

0-8186-0554-5
CATALOG NO. 84CH2004-0
LIBRARY OF CONGRESS NO. 83-83444
COMPUTER SOCIETY ORDER NO.554

PROCEEDINGS
SEVENTH ANNUAL

MICRO-DELCON '84

THE
DELAWARE BAY
COMPUTER
CONFERENCE
1984



Sponsored by
IEEE
Computer Society
Delaware Bay
Section



INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS

MARCH 6, 1984
John M. Clayton Hall
University of Delaware
Newark, Delaware

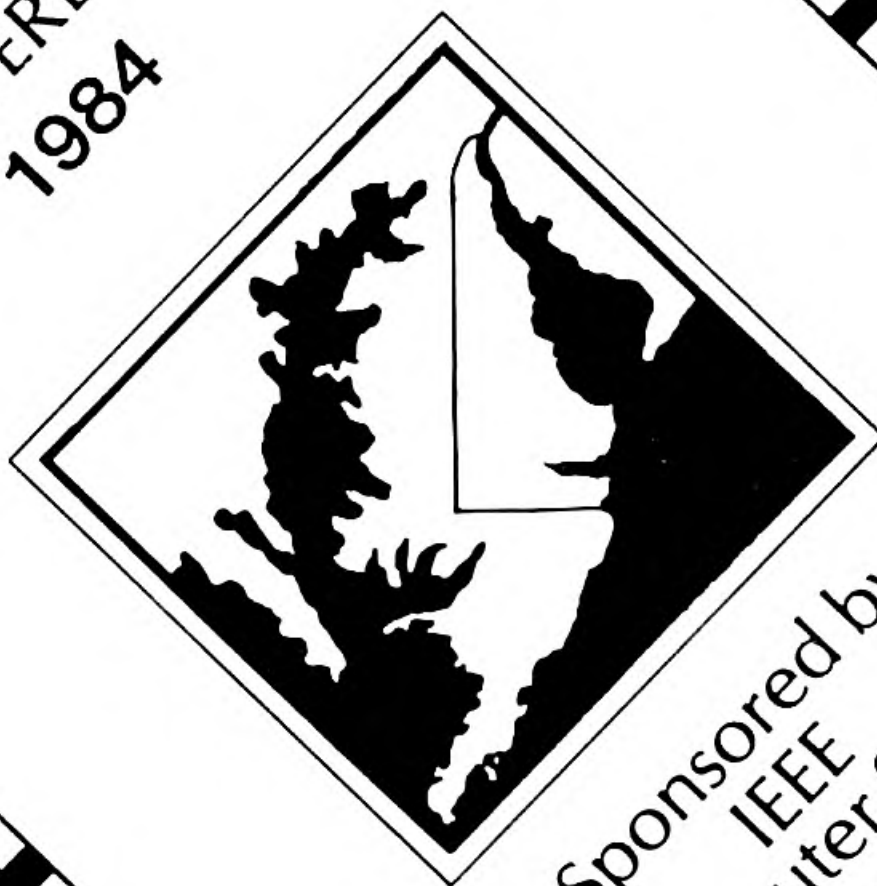
IEEE
COMPUTER
SOCIETY
PRESS 

ISBN 0-8186-0554-5
IEEE CATALOG NO. 84CH2004-0
LIBRARY OF CONGRESS NO. 83-83444
IEEE COMPUTER SOCIETY ORDER NO.554

PROCEEDINGS
SEVENTH ANNUAL

MICRO-DELCON '84

THE
DELAWARE BAY
COMPUTER
CONFERENCE
1984



Sponsored by
IEEE
Computer Society
Delaware Bay
Section



INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS

MARCH 6, 1984
John M. Clayton
University of Delaware
Newark, Delaware

THE COMPUTER
SOCIETY
PRESS

The papers appearing in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and are published as presented and without change, in the interests of timely dissemination. Their inclusion in this publication does not necessarily constitute endorsement by the editors, IEEE Computer Society Press, or the Institute of Electrical and Electronics Engineers, Inc.

Published by IEEE Computer Society Press
1109 Spring Street
Suite 300
Silver Spring, MD 20910

Copyright and Reprint Permissions: Abstracting is permitted with credit to the source. Libraries are permitted to photocopy beyond the limits of U.S. copyright law for private use of patrons those articles in this volume that carry a code at the bottom of the first page, provided the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 29 Congress Street, Salem, MA 01970. Instructors are permitted to photocopy isolated articles for noncommercial classroom use without fee. For other copying, reprint or republication permission, write to Director, Publishing Services, IEEE, 345 E. 47 St., New York, NY 10017. All rights reserved. Copyright © 1984 by The Institute of Electrical and Electronics Engineers, Inc.

ISBN 0-8186-0554-5 (paper)
ISBN 0-8186-4554-7 (microfiche)
ISBN 0-8186-8554-9 (casebound)
IEEE Catalog No. 84CH2004-0
Library of Congress No. 83-83444
IEEE Computer Society Order No. 554

Order from: IEEE Computer Society
Post Office Box 80452
Worldway Postal Center
Los Angeles, CA 90080

IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854



The Institute of Electrical and Electronics Engineers, Inc.

FOREWORD

MICRO-DELCON '84 is the Seventh Annual Conference sponsored by the Computer Society of the Delaware Bay Section of the IEEE. The meeting provides a one-day forum at which professionals from all areas related to computer science and computer engineering can exchange ideas.

The goal of our annual conference is to promote interaction among academic and industrial professionals in the Delaware Bay area. Traditionally, each of these sources is responsible for roughly one-half of the papers presented. In the last few years, the conference has begun to attract interest well outside of our local area, so that an area within a 200-mile radius of Newark is included.

We are fortunate to have two excellent keynote speakers. We are grateful to Dr. Henry Sowizral from the Rand Corporation (who is discussing the control of control) and to Dr. John Hutchison from the General Electric Corporation who is giving an up-to-date look at ADA.

The success of any conference is due to the authors and the referees; we gratefully acknowledge their contribution. The conference would not exist without the voluntary labor of all the committee members. We are also indebted to the staffs of the Departments of Computer and Information Sciences and of Electrical Engineering of the University of Delaware, whose help was essential to the success of the conference.

John T. Lund
Conference Chairman

A. Toni Cohen
Technical Program Chairman

MICRO-DELCON '84

SEVENTH ANNUAL CONFERENCE ON COMPUTER TECHNOLOGY

March 6, 1984

John M Clayton Hall - University of Delaware - Newark, Delaware 19711

CONFERENCE COMMITTEE

Conference Chair
John T Lund

Technical Program
A Toni Cohen

Conference Vice Chair
Bobby F Caviness

Finance
Janet Lund

Exhibits
Charles R Berg

Registration
David M Robinson

Publicity
John S DeGood
Mark E Laubach

Arrangements
W Barkley Fritz

Membership
Nancy Carey

PROGRAM COMMITTEE

Gonzalo Arce

Tim Finin

Allen Haley

Joseph E Lambert

Roger A Patterson

Donald Perlis

MICRO-DELCON '84

Table of Contents

Keynote address: *The Control of Control*
Dr. Henry Sowizral (Rand Corporation)

Session I: High Level Software
Chair: Allen Haley

<i>A Database Construction Kit</i>	1
Mark B Reinhold	
<i>Structure-level Specification: Program Design</i>	11
Margaret J Davis and A Toni Cohen	
<i>Odradek: A Prolog-based Lisp Translator</i>	18
Herb Jellinek	
<i>Rule-based Computer Alarm Analysis in Chemical Process Plants</i>	22
Daniel Chester, David Lamb and Prasad Dhurjati	
<i>Mutation Analysis: A User's View</i>	30
James M Galvin	

Keynote address: *An Overview of Ada*
Dr. John Hutchison (General Electric Company)

Session II: System Control
Chair: Gonzalo Arce

<i>Computer Requirements for Nontraditional Control</i>	41
Ronald K Pearson	
<i>A Static Multivariable Feedforward Compensator</i>	48
Michael J Piovoso and Ronald K Pearson	

Session III: Hardware System Implementations

Chair: Tuncay Saydem

<i>INUZ - A Project for Interconnecting Computer Networks</i>	61
Zhigang Fan	
<i>The Wang Professional Image Computer: An Overview</i>	68
Ahmed El-Sherbini and Mike Smutek	
<i>A Comparison of Two Communication Networks for Videotex Systems</i>	74
AW Mansky	

Session IV: Tightly-coupled Parallel Architectures

Chair: Roger Patterson

<i>A Data-stationary Pipelined Machine</i>	79
Ikram E Abdou	
<i>Data Structures on Dataflow Computers: Implementations and Problems</i>	84
Susan Samet and Maya Gokhale	
<i>Hardware Architecture of the Parallel Finite State Model</i>	97
AR Hurson and B Shirazi	
<i>Evaluation of the Manchester Dataflow Machine</i>	105
Maya Gokhale, Lance Ramshaw and Ioli Constantinou	

Session V: Digital Signal Processing

Chair: David M Robinson

<i>Recursive 2 Dimensional Median Filtered Signals with Impulsive Noise</i>	117
RJ Crinon and GR Arce	
<i>A Median Filter Architecture Suitable for VLSI Implementation</i>	124
Peter J Warter, Gary S Delp, Jorge L Garcia-Colevatti, John Dalton, John Baumeister	
<i>Median Filters as Spectral Windows</i>	131
RK Pearson and GR Arce	

AUTHOR INDEX

Abdou, I	79	Garcia-Colevatti, J	124
Arce, G	117, 131	Gokhale, M	84, 105
Baumeister, J	124	Hurson, A	97
Chester, D	22	Jellinek, H	18
Cohen, T	11	Lamb, D	22
Constantinou, L	105	Mansky, A	74
Crinon, R	117	Pearson, R	41, 48, 131
Dalton, J	124	Pivoso, M	48
Davis, M	11	Ramshaw, L	105
Delp, G	124	Reinhold, M	1
Dhurjati, P	22	Samet, S	84
El-Sherbini, A	68	Shirazi, B	97
Fan, Z	61	Smutek, M	68
Galvin, J	30	Warter, P	124

A DATABASE CONSTRUCTION KIT

Mark B. Reinhold

University of Pennsylvania

Abstract

A programmer building a database is often faced with the choice between implementing the storage structures himself or using a general-purpose database management system. The former requires considerable effort. The latter may be even less desirable because DBMSs are expensive and unwieldy; moreover, they impose a data model on the programmer which may not suit the problem at hand. A Database Construction Kit has been designed and implemented that provides modular access to many techniques for the storage, indexing and retrieval of data. It is intended to be used in both small application programs only needing a random-access file, as well as in the implementation of complete personal database management systems.

Introduction

The Database Construction Kit is a file system specifically designed for use by applications which engage in any sort of database-like activity, be it simple random file access or the implementation of a database management system for a new data model. It provides a practical alternative to the usual choices of constructing one's own database mechanism or using a general-purpose database management system. The following principles were important in the design of the Database Kit.

- **Structural Extensibility.** An initial application of the kit will be to build a database management system for the functional data model. Central to the functional data model is the concept of the separate storage of data domains. To allow an efficient implementation of the functional model, such domains have been made inexpensive to create and destroy.
- **Reliability.** There is wide variation in the reliability of the file systems provided by general purpose operating systems. The kit does everything in its power to improve a database file's chances of surviving a machine crash, a power failure, or other minor disaster. Obviously, there are some events (e.g., a disk head crash) beyond the power of mere software.

- **Flexibility.** If some application requires a special-purpose storage or index mechanism, it can be integrated into the file system. If a particular application does not intend to construct B-tree indices, it does not have to carry around the module that implements B-trees.
- **Portability.** A significant amount of effort is involved in writing a file system, even a simple one which leaves the details of I/O to the host operating system. The code was written in as portable a style as possible, and no assumptions were made about esoteric capabilities of the host operating system.
- **File Access Efficiency.** Both the software and the file structure are designed to minimize the number of I/O operations required for any given task.

Concurrent database access is not provided. While the problem of synchronizing concurrent database accessors so that each sees a consistent view of the database is well understood⁷, the details of locking and shared file access vary so much from system to system that it is difficult to define a simple file sharing and locking mechanism that could be implemented on all systems. The kit is targeted for small but sophisticated "personal" databases and application programs for which shared access is not required. If absolutely necessary, concurrent access to a database could be provided by having all accessors interact with the file system through a common server process.

Database Abstractions

The Database Kit provides two low-level abstractions—*stores* and *indices*—from which an application builds its own higher-level structures. A store has the ability to:

- Save a datum and return an identifier marking its location;
- Retrieve a datum given its identifier;
- Update a datum in place, such that it is still marked by the same identifier;
- Delete a datum from storage.

Rather than operating on arbitrary data, an index deals with (*key*, *value*) pairs in the following ways:

- Insert a (*key*, *value*) pair into an index;
- Retrieve the *value*(s) corresponding to a given *key*;
- Delete a (*key*, *value*) pair from its index once it has been retrieved.

A database is a collection of stores and indices. A *storage method* is the program module which implements a particular kind of store; an *index method* is the module which implements an index. Collectively, the modules which implement stores and indices are referred to as the *access methods*.

Stores and indices are built within *segments*. A segment is a logical partition of the database; a different segment is used for each store or index. When a store or index is created, a *segment number* is returned which denotes the segment in which the structure resides. The segment number is used from that time onward to refer to the structure until it—or the database—is destroyed.

Stores

The simplest type of store is that for fixed length records. Such stores are easy to implement, but many types of data (e.g., character strings) come in varying sizes and should be stored so as not to waste space. Thus there are two types of stores: one for fixed-length records (record storage) and one for variable-length records (text storage).

The following operations are defined on stores.

create(*type*, ...) → *seg*

Create a new store; *type* denotes the type of store to be created. Additional parameters may specify store-type-specific information such as the record length or the expected average character string length. *create* returns the segment number of the new store.

destroy(*seg*)

Destroy the store residing in segment *seg*.

put(*val*, *seg*) → *id*

Store the value *val* in segment *seg*, returning a storage identifier with which it can subsequently be retrieved.

get(*seg*, *id*) → *val*

Retrieve the value denoted by identifier *id* in segment *seg*.

delete(*seg*, *id*)

Delete the value denoted by identifier *id* in segment *seg*.

update(*val*, *seg*, *id*)

Replace the value denoted by identifier *id* in segment *seg* with the new value *val*.

The storage identifiers associated with stored values are *magic*. They may be freely passed around, stored away in other stores or indices, or even converted to hexadecimal and printed to the terminal, but no interpretation whatsoever may be placed upon them. This policy ensures that no application will take advantage of knowledge about the implementation of a storage method, thereby becoming dependent upon that particular implementation of that particular storage method. The single exception to this rule is that storage identifiers from the same store may be compared with one another. The property that all storage identifiers for a given store are *unique* is often useful when composing larger structures.

Indices

There are more types of indices than stores. The present implementation offers B-trees² with modifications as suggested by Knuth¹⁰, extendible hash indices⁶, and associative string spaces. The type of index to use for a given application is determined by the type of the key data, the way in which the (*key*, *value*) pairs are to be retrieved, and the space versus time tradeoff. The space-time tradeoff is often the most important; generally, the faster an index can be searched, the more space it occupies and the more space it leaves unused.

In order to build an index and retrieve data from it, an index method must have some way of interpreting key data so that keys can be compared; storage methods have no need to interpret the data they handle. All data passed between the application and the file system for storage or retrieval purposes are in the form of *byte strings*. Byte strings are dynamically typed; i.e., part of their value denotes the type of the data in the string so that it may be compared with other strings of the same type. The other components of a byte string's value give the length of the string and a pointer to the string itself.

Index methods differ in how they allow (*key*, *value*) pairs to be inserted. There are three possibilities for the *insertion mode* of an index:

- Duplicate keys are not allowed;
- Duplicate keys are allowed; or
- Duplicate keys are to be overwritten.

In the last case duplicate keys are not allowed, and inserting a (*key*, *value*) pair whose *key* is already in the index will result in the automatic deletion of the old pair. Some indices support all insertion modes; some allow only one.

The insertion mode of an index is specified when the index is created, along with the type of the keys of the index and the type of the index itself.

create(*type*, *mode*, *keytype*, ...) $\rightarrow$ *seg*

Create an index. *type*, *mode*, and *keytype* denote the type of the index, its insertion mode, and the type of keys in the index, respectively. Additional parameters may specify index-type-specific information such as the maximum key length.

Unlike operations on a store, some of the operations on an index must retain state information from one call to the next. Furthermore, most of these same operations must return a value indicating the success or failure of the file system to service the request. *Cursors* solve both of these problems. A cursor has two components: a *status* and a *position*. The *status* component indicates the success or failure of an operation and provides some additional status information. The *position* component marks the position of a (*key*, *value*) pair in an index. Applications may access the status value, but—like storage identifiers—the position information in a cursor is magic. Some operations such as *insert* return cursors only to indicate a status; the position part of such a cursor is meaningless.

insert(*key*, *val*, *seg*) $\rightarrow$ *cursor*

Insert the given (*key*, *value*) pair into the index in segment *seg*. The returned *cursor* reports the status of the insertion operation. The type of *key* must agree with the key type given when the index was created. The type of *val* is irrelevant.

A second distinctive characteristic of the index methods is the manner in which keys may be retrieved. The key being searched for is referred to as the *target key*. All index methods allow a *literal match*: if a retrieve operation is performed for a target key which exactly matches a key in the index, the search is successful and the corresponding (*key*, *value*) pair is returned. The initial retrieve operation is named *first*, since it returns the first matching pair.

first(*seg*, *key*) $\rightarrow$ *key* $\times$ *val* $\times$ *cursor*

Find the first (*key*, *value*) pair in segment *seg* whose key matches the target *key*. The type of *key* must agree with the key type given when the index was created. The returned triplet contains the actual key (since the match may not have been exact—see below), the corresponding value, and a cursor which reports the status of the retrieval and marks the position of the (*key*, *value*) pair in the index for purposes of *delete* or *next*.

If there is more than one matching pair, the remaining pairs may be obtained by means of the *next* operation.

next(*cursor*, *key*) $\rightarrow$ *key* $\times$ *val* $\times$ *cursor*

Find the (*key*, *value*) pair whose key matches the target *key* and which follows the pair whose position is marked by the given *cursor*.

A table is an example of a literal-match index. A table is an array: the key of a pair is the unsigned integer index of the value in the array. Table indices do not allow duplicate keys, since there can be only one array element for a given array index.

There are two variations on the literal match: the *prefix match* and the *pattern match*. A prefix match is successful if a key in the index has the target key for a prefix. It is possible that a key is exactly equal the target key; the status value of the cursor is used to distinguish between exact and prefix matches. Any additional keys that match the prefix may be obtained with *next*. If there are a number of exact matches followed by a number of prefix matches, the exact matches are returned first. A B-tree is a prefix-match index.

A pattern match allows a search for some pattern. A pattern defines a (possibly infinite) set of strings, which set the key must be a member of in order for the match to be successful. A set of strings which denotes the set of all strings containing those strings is a simple kind of pattern; a regular expression¹⁸ is a more powerful kind of pattern. Pattern-matching indices allow rich and complex key specifications at the expense of being much slower than other index methods. An example of a pattern-matching index is an *associative string space*, which stores (*key*, *value*) pairs in random order and retrieves them by scanning the index from beginning to end.

Together, the *first* and *next* functions can be used to retrieve the sequence of (*key*, *value*) pairs whose keys match a target key, but nothing has been said about the order of that sequence. The *natural order* of an index is defined as an ordering of the (*key*, *value*) pairs of an index which is some function of the order in which the pairs were inserted into that index. The natural order may be by key value, as in a B-tree, or it may be random, as in a hash index that uses a non-order-preserving hash function. When a sequence of (*key*, *value*) pairs is obtained from an index with *first* and *next*, the order of the sequence is the natural order of the index.

An index may be completely traversed by using a null key. By proclamation, a null key will always match any key in an index, so performing a *first* operation with a null target key should give the very first (*key*, *value*) pair in the index. Then *next* operations can be performed, using the same null key, until the end of the index is reached. A complete traversal yields all the (*key*, *value*) pairs in the index in their natural order.

A (*key*, *value*) pair may be deleted using the cursor returned by *first* or *next*.

delete(cursor) → cursor

Delete the (*key*, *value*) pair denoted by *cursor*.

In general, a *delete* operation implicitly invalidates all cursors on the index to which it is applied, including the cursor returned by *delete*. However, some index methods allow a *delete* to be followed by a *next* on the resultant cursor. All other cursors become invalid, but the cursor returned by *delete* is adjusted to reflect any changes in the internal structure of the index. The status value of a cursor returned by *delete* is irrelevant; delete operations either succeed or abort.

Not all indices fit neatly into this model. Therefore a *special* function is introduced which is used as an escape for those operations not yet described.

special(seg, request, ...) → cursor

Perform some special operation, denoted by *request*, on the index in segment *seg*. The value of the returned *cursor* may or may not be meaningful.

One use of the *special* function is with a table index. In some applications the actual position of an element in the array is unimportant, but all the used positions must be packed down at the bottom of the array, i.e., sequentially numbered from 0 to *n* with no gaps. When inserting a new element, the key of the first unused position is found with the *special* function and then used as the key for the insertion operation.

Databases

Finally, there are functions to manipulate the entire database. Initially a database must be created or opened.

create(name) → fullname

Create a new database with the given *name*, returning its full name.

open(name, mode) → fullname

Open an existing database of the given *name*, returning its full name. *mode* indicates whether the open is for read-only or for read/write operations.

When the various storage and index methods are called to update the database, the database file is not actually modified until the current transaction is committed. A transaction is a set of database operations terminated by *commit*. The file system does the work of guaranteeing that transactions are *atomic*; an atomic transaction is one that either runs to completion or has no effect on the database at all¹¹. Until a transaction is committed, there is no visible record in the database file that it was ever

started, so if the system suddenly crashes, the database is left in a consistent state.

commit

Declare the end of a transaction. When *commit* returns, any changes to the database have been irrevocably committed.

A transaction may be *aborted* at any time before the commit point, either by the application (e.g., in case of operator error), by the database file system itself (e.g., in case of a shortage of some resource such as disk pages), or inadvertently (e.g., in case of a machine crash). Of course in the latter case, nothing will be around to call the *abort* function. All of the functions introduced in this section will trigger an internal abort if something goes wrong unexpectedly.

abort

Abort the current transaction. Any pending database updates are thrown away and forgotten.

When the application is finished with the database, the database file should be closed.

close

Close the currently open database.

Using the Kit

In order to provide a more intuitive feel for how the Database Kit is used, this section discusses how one could implement database management systems for the functional, relational, and network data models using the database abstractions just presented.

The Functional Data Model

The functional data model describes information in terms of entities and functions that define mappings between entities¹⁶. For example, in a university database a particular student and the courses he is taking may be modelled as a function mapping a student entity to a set of course entities.

Functions exist in several varieties. Functions may be defined with zero or more parameters, and functions may return a single value or a (possibly empty) set of values. A function that returns a singleton is expressed with a single arrow ($\rightarrow$), while a function that returns a set is expressed with a double arrow ($\Rightarrow$).

A special function is defined for each type of entity: the *generator function* for an entity type has no parameters and returns a set containing all the entities of that type in the database.

Here are some sample functions on the university database:

```
person()  $\Rightarrow$  entity
name(person)  $\rightarrow$  string
course(student)  $\Rightarrow$  course
grade(course, student)  $\rightarrow$  integer
```

The database must contain some metadata—data describing the structure of the rest of the data—so that the functional DBMS can figure out how to evaluate functions. For each function, a *function descriptor* contains the following items: how the function is stored, what segment(s) it is stored in, the number of parameters it takes and their types, the result type of the function, and whether the function returns a single value or a set. Two segments are used for the metadata. A record store contains the function descriptors, and an extendible hash index maps function name strings into the storage identifier of the corresponding function descriptor.

The type of an entity is denoted by the storage identifier of the function descriptor for its generator function. Each entity within a type is assigned an instance number unique to that entity type. Generator functions are stored in a table index whose value size is zero; the table is used only to uniquely identify entities—by their instance numbers—and to traverse that set of instance numbers. A table with a zero value size is just a bitmap, which is a very efficient way of storing a generator function. When a new entity is created, the key of an empty table position is obtained with the *special* function.

A 1-to-1 function is stored in a table whose key is the domain and whose value is the range. A 1-to- n function is stored in two segments. A text store holds the range set of the function; a table maps the domain entity instance number to the storage identifier of the corresponding range set. An n -to-1 function is stored in an associative string index with the domain tuple as the key; a table would be inappropriate here as such functions tend to be very sparse. If fast access were absolutely necessary, a B-tree or extendible hash index could be used instead. n -to- n functions are implemented in a manner similar to 1-to- n functions, except that an associative string index is used instead of a table to map domain tuples into range set storage identifiers.

Only small values such as entity numbers and integers can be directly passed around as function values. Character strings are treated in the following way. A global string space is maintained in a text segment and in a B-tree; the B-tree maps strings into string numbers, and the text segment maps string numbers back into strings. Functions dealing with strings refer to strings in these segments. This approach would be duplicated for any other data types with large values.

The Relational Data Model

See Date⁴ for an introduction to the relational data model. Assume for simplicity that tuples in a relation are of fixed length. A single relation may be stored in a table index, one tuple per record, where the key of each record is its row number in the relation. An index is used here instead of a store because it allows an efficient traversal of the rows of the relation. An index on a relation can be constructed in a B-tree or extendible hash segment. Key uniqueness may be enforced by creating the index to not allow duplicate keys.

The metadata required for a relation includes the number of the table segment containing the relation, the number of fields in each tuple and the type and length of each field, and any information required to link a relation with its associated indices. A descriptor record for each relation is kept in a record store, while an extendible hash index provides a mapping from relation name to relation descriptor record storage identifier. The descriptor record may be extended to describe any integrity constraints applicable to the relation.

The Network Data Model

Again, Date⁴ provides a readable presentation of the network (CODASYL) data model. A simple implementation of the network data model would use a record store for each record type and then construct the traditional pointer chains from record to record within each set. However, a design which takes more advantage of the file system's capabilities is possible.

A record store is used for each record type. A table index is used to store descriptor records for each record type containing the number of the record segment, the type and length of each field, and any constraints that apply to the record (e.g., a field must be unique within a set).

A set is implemented by means of two segments. A table index maps an owner record number to a member set number, and a text segment maps the member set number to a set, which contains zero or more member record storage identifiers. Since record numbers only identify records within a segment, a schema table must be kept describing each set in terms of its owner record type, member record type, and the numbers of the segments containing the owner-set table and the sets themselves. Indices that provide fast access from key to record may be constructed on B-trees or extendible hash segments as necessary.

A Real Application

The first application of the Database Kit was to construct a program to maintain a database of bibliographic refer-

ences in the style of REFER¹³; that program was used to prepare the references for this paper.

The structure of the bibliographic database is as follows. Each entry consists of one or more fields of text (author, title, date, etc.). The keys for an entry are concocted from all words in that entry; keys which are numbers outside of the range 1800-2000 (i.e., not viable year numbers) are ignored, as are keys which are on a list of the 100 most common English words. All keys are stored in lower case and are truncated to twenty characters. A database contains the following segments:

<i>entid</i> → <i>entry</i>	text store; contains actual entries
<i>entsetid</i> → { <i>entid</i> }	text store; contains sets of entry ids
<i>key</i> → <i>entsetid</i>	B-tree; maps key to entry set id
<i>0</i> → <i>entid</i>	table of all entries

Entries are kept in a text store. Since a key may belong to more than one entry, the value of a key in the key index is the storage identifier of a text string containing one or more storage identifiers for the entry store. A table is used to keep track of all entry identifiers so that all the entries in the database can be produced, e.g., in order to unload the database into a text file.

A bibliographic citation is specified by giving a set of keys for the entry. For example, volume one of Knuth's *The Art of Computer Programming* series could be cited with the keys "Knuth Art Computer 1973." Each key is retrieved from the *key* → *entsetid* index; each of these *entsetids* produces a set of *entids*, the intersection of which produces the set of storage identifiers corresponding to the desired references. The file system performs quite well in this application; it takes approximately 0.15 CPU seconds and only 9 disk reads for the program look up all the references in this paper from a database of about 100 entries.

Implementation Overview

This section briefly describes how the implementation of the Database Kit meets its design goals. A soon-to-appear technical report¹⁵ gives extensive details on the implementation.

Software Structure

The initial implementation of the database kit was done on a VAX under the VMS operating system and the *Eunice* UNIX* emulator⁸. It is written entirely in the C programming language⁹ according to the UNIX version 7 standard. The implementation depends upon no special features of the VAX, so it could be ported to a PDP-11 with minimal effort. High quality compilers are now becoming available for larger microcomputers (e.g. 68000-based systems), extending the set of machines to which this implementation could be transferred. Of course, it will run on any VAX with DEC's VMS operating system.

*UNIX is a trademark of Bell Laboratories.

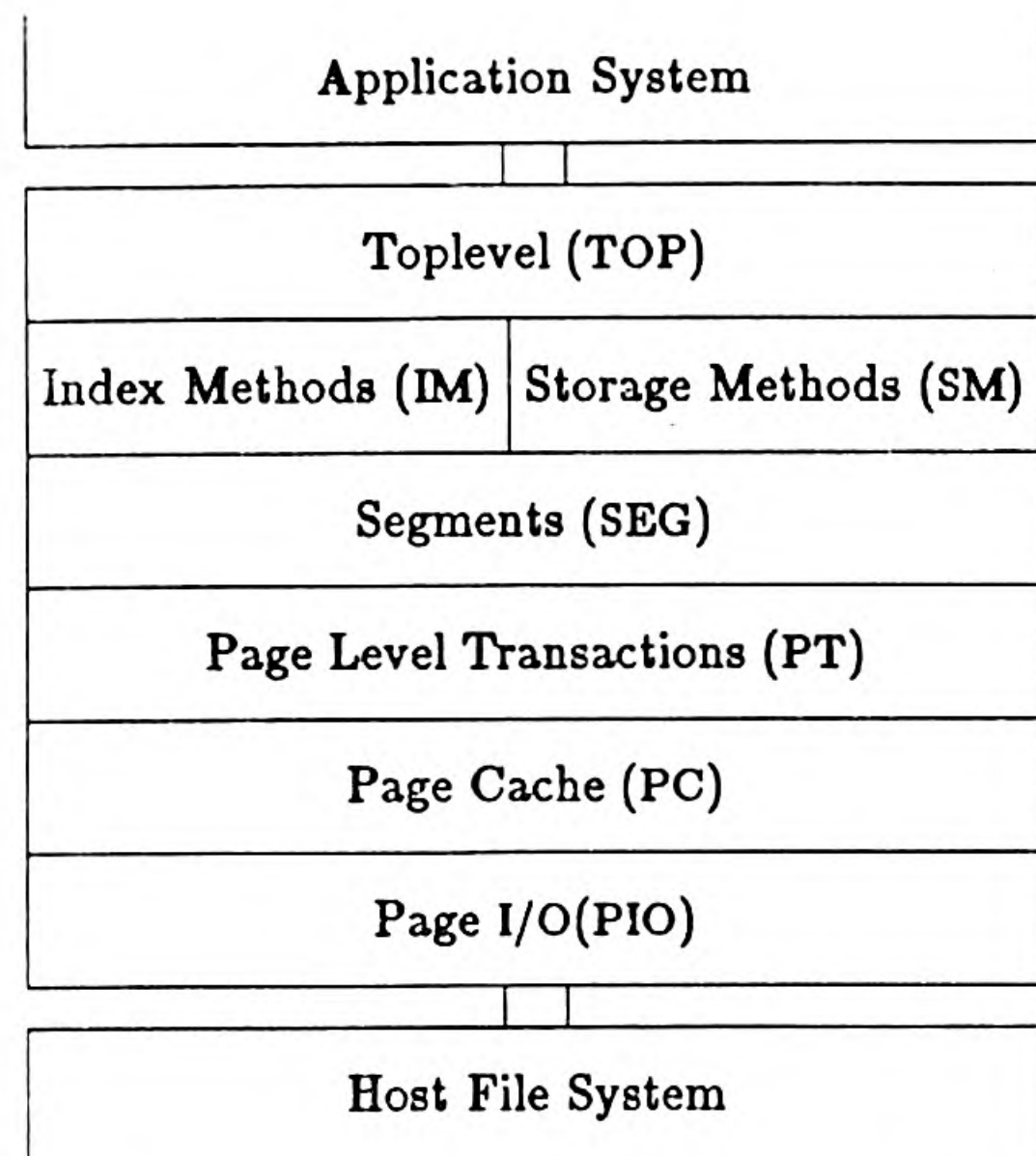


Figure 1. Software Layer Structure

The file system is organized as a set of layers, one on top of the other. Each layer provides some abstraction in the form of a set of functions and procedures to the layer above it, and uses the abstraction provided by the layer below it to construct its own abstraction. Externally, a layer is defined only by the services it implements and the services it uses; the implementation of the layer is completely invisible to the outside world. The independence of interface and implementation allows different versions of a layer to be used in different versions of the file system. Figure 1 depicts the overall structure of the file system.

The PIO layer uses the host's file system to provide the abstraction of a file of pages numbered from zero. Under VMS, this is a fairly direct mapping to the file I/O services of the RMS record management services component⁵. To maximize portability, all that is assumed of the host operating system is that it can write disk pages atomically, i.e., a write operation either completes successfully or does nothing at all.

The purpose of the page cache layer is to reduce file activity by keeping the most frequently referenced database pages in memory. The abstraction provided by the PC layer is that of a set of *page frames*, each of which contains a single page of the database file. The number of page frames may be varied by means of a compile-time constant. The PC layer uses a modified least-recently-used (LRU) algorithm to select which pages are to be kept in the cache. The LRU algorithm takes advantage of additional knowledge about how a page frame is intended to be used, as suggested by Stonebraker¹⁷. When a page is requested, information such as the manner in which the caller intends to access the page in the near future and

whether it will be written or not is used to determine the length of time for which the page will remain in the cache.

The page-level transaction layer handles the database update process that takes place during a transaction and is finished up at transaction commit. It is also responsible for maintaining the freelist—the list of unused pages in the database file—during a transaction. Further discussion of this layer requires knowledge of the file structure, and will be postponed until the next subsection.

The segment layer uses the pages provided by the PT layer to build database segments. A segment is a set of pages arranged in a linear address space that is not necessarily contiguous; i.e., some page numbers may not map to valid disk pages. Each page contains a fixed number of bytes. The storage identifier for a particular byte is composed of the number of the page within the segment and the offset of the byte in that page. Each segment has a *type* attribute which describes the type of structure it contains. This model is much like that of the Relational Storage System used in the System R relational database management system¹.

The access method layers, IM and SM, use the segments provided by the SEG layer to build stores and indices. Unlike most layers of the system, the IM/SM layer consists of several modules. One module is present for each access method in use; two additional modules contain supporting code for segment space management and other utilities such as byte string comparison and hashing.

The outside world only calls functions in the toplevel. The functions available to the application program correspond directly to those outlined in the section on database abstractions. The functions for manipulating indices and

stores are overloaded: there is one set of functions for all operations upon stores and one set of functions for all operations upon indices. For example, there is a single function to delete a (*key, value*) pair from an index given a cursor obtained from *first* or *next*, which function is used on each and every index when that operation is required. The toplevel layer determines which access method module to call by examining the *type* attribute of the segment in question.

The toplevel layer is also responsible for handling and reporting transaction aborts. If the host operating system provides the required services, the toplevel layer protects against internal errors in the file system code by catching system error traps and using them to initiate internal aborts. Information about a transaction abort is made available to the application through two global variables and a user error trap function.

File Structure

In order to implement efficient and independent segments, simple atomic database updates, and to take advantage of locality of reference within a segment, the database file is structured as a tree (see figure 2). At the lowest level (the PIO layer) the file appears as a collection of sequentially numbered pages. The next two layers of the file system (PC and PT) transform this view into a tree of pages, with the zeroth page as the root. The root page points to a set of subtrees, each of which constitutes a segment. The root of each segment subtree is the *segment header*, which contains the number of pages allocated to the segment, the type of the structure stored in the segment, and direct and indirect mapping pointers to the data pages in the segment.

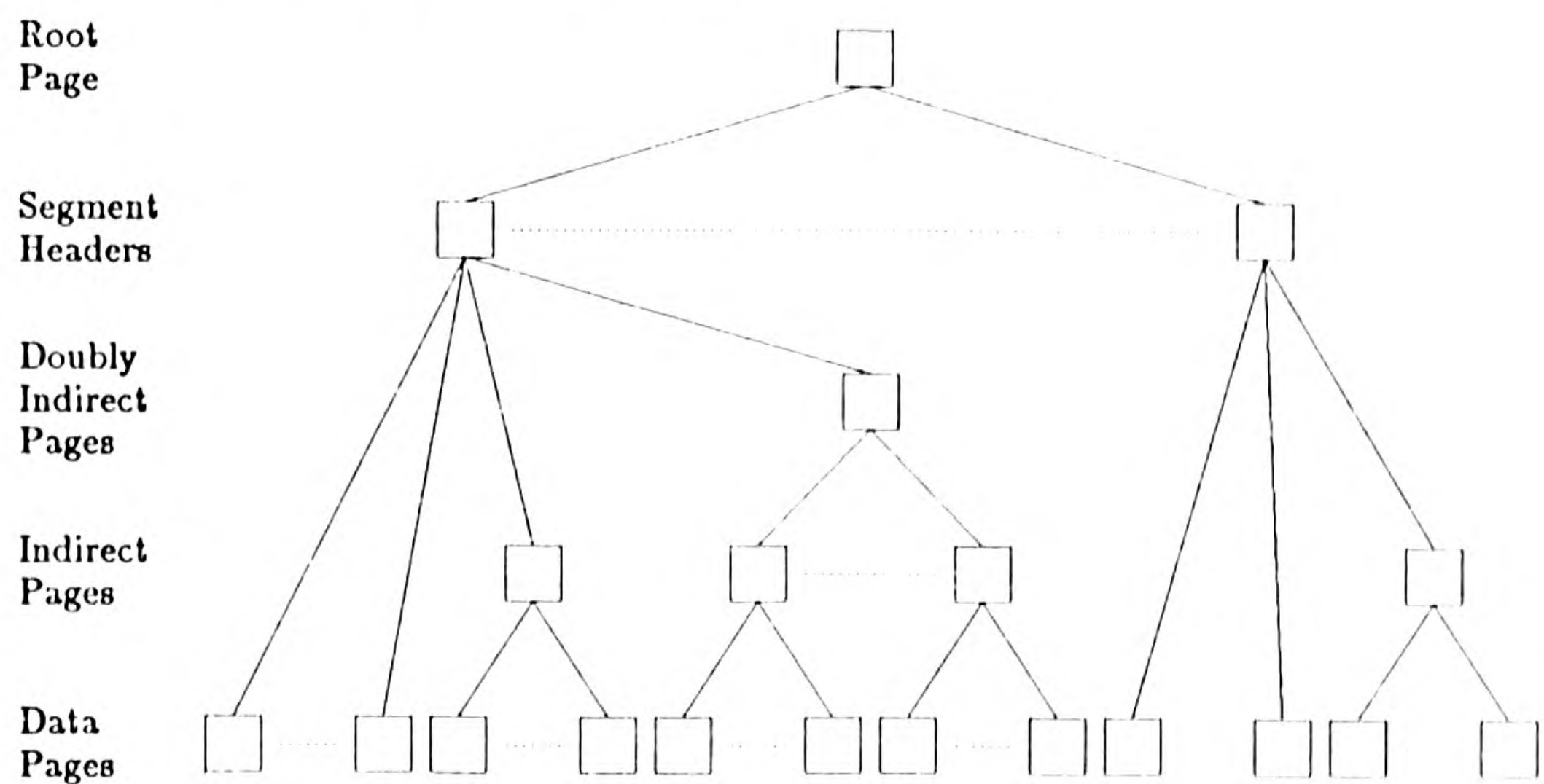


Figure 2. Database File Structure

Several attributes of the file structure are variable and may be changed by means of compile-time constants. This allows a certain amount of "fiddling" to tune the system for particular applications. The most important attribute is the file page size, which affects all other file structure constants. Pages are typically between 2^9 and 2^{12} bytes long and are required to be integer multiples of the disk page size of the host operating system. No part of the file system depends upon a particular page size; the current implementation has been verified to work with page sizes of 512, 1024, and 2048 bytes. Pages within a segment are denoted by *segment page numbers*, which are typically two or three bytes long, allowing from 2^{16} to 2^{24} pages in a segment. The number of pages denotable by a segment page number may be smaller or larger than the structural limit of the segment dictated by the page size constant.

Let s be the file page size measured in units equal to the size of a segment page number. If i units are required to store other information, then the segment header consists of $s - i$ mapping pointers. All but two of the mapping pointers point directly to data pages; thus the first $s - i - 2$ data pages may be accessed directly from the segment header page. The segment header points to one indirect mapping page, which points to s data pages; the segment header also points to a doubly indirect mapping page, which points to s indirect mapping pages, each of which in turn points to s data pages. In terms of the page size s , a given segment may contain $s^2 + 2s - i - 2$ data pages. For a page size of 1024 bytes and a segment page number size of two bytes, $s = 512$, so if two units are taken up for other information ($i = 2$), the segment capacity is 263164 pages (269 megabytes). In this case, the structural limit is greater than the limit imposed by the size of the segment page number, so each segment in fact would be limited to 65535 data pages, or 67 megabytes.

Since segments are naturally subtrees of the tree structure of a database file, they incur little extra overhead of themselves. It is not unreasonable to consider a database of hundreds of small segments, each containing a single data domain for some data model. Segments can be created and destroyed by the SEG layer independently of the data structures built within them. This separates segments from the structures they contain and isolates each segment from all others in the database, reinforcing one's faith in the file system. A bug in the module that implements B-tree indices may cause some random page to be overwritten, but only in the segment containing the faulty B-tree; other segments will be left alone.

The tree-based file structure helps reduce file activity by decreasing the access cost for a page as locality of reference within a segment increases. n references to a small area within a segment are cheaper than n widely spaced

references in a segment, which in turn are much cheaper than n references to entirely different segments. There is also a bias in favor of segments whose data is clustered at the bottom end: references to the first $s - i - 2$ pages in a segment will be cheaper than references to the next s pages, which in turn are cheaper than references to the remaining s^2 pages. This is due to the intentionally slightly lopsided mapping scheme described above.

The index and storage methods can improve locality of reference through the use of *near-mode allocation*. When a new page is created in a segment, the caller can specify the page number of an existing page. The SEG layer tries—but is not guaranteed—to allocate the new page as near as possible to the old page. Near-mode allocation only works in terms of the internal segment structure; it does not attempt to provide physical contiguity in the disk file.

Another advantage of the tree-based file structure is that it admits a simple algorithm for performing completely atomic updates. When a segment page numbered d is to be written, a function in the SEG layer recursively descends the tree, beginning at the root. When page d is reached, a call to the PT layer finds an unused file page for it and writes it to that page through the cache. The PT layer returns the number p of the file page assigned to the segment page. The pointer to the old copy of d in the page immediately above is modified to point to its new location in the file, namely the file page numbered p .

The recursion unravels until the root page is reached. The appropriate pointer in the root page is modified, but the root page is not written until the transaction is committed. Since disk write operations are assumed to be atomic, the final action of writing the root page—always back to file page zero—updates the entire file in a single atomic operation. Until the modified root page is written, all newly written pages are still in the freelist. If a transaction aborts before the root page is written, the database is left in a consistent state. This scheme was inspired by Paxton's transaction mechanism for distributed file systems¹⁴.

Just before the root page is written to commit the transaction, the freelist is updated. As pages are allocated and freed during a transaction, a list of *recently freed* pages is built using pages still in the freelist. A separate list of recently freed pages is required because a page freed during a transaction cannot be re-used in the same transaction. If it were rewritten and the transaction were aborted then the database would not be left in a consistent state. At commit time, the recently freed page list is merged with the old free page list to form a single new freelist. Because the freelist update takes place immediately after the rest

Table 1. Index Method Performance

Index Method	A-string	B-tree	ISAM
Construction			
disk writes	1512	5023	5082
CPU time	67.95	113.17	32.80
page faults	597	578	343
file size	228	322	396
space utilization	96.9%	70.1%	46%
Retrieval			
disk reads	109	118	180
CPU time	70.71	2.61	4.63
page faults	38	45	5
cache hit ratio	99.7%	95.7%	—

of the file pages are updated and just before the root page is written to commit the transaction, aborting the transaction will cause the file pages containing the recently freed page list to remain in the old freelist.

Performance

Each storage method was tested by randomly executing put, get, update, and delete operations on a large fixed quantity of data. Both the record and text storage methods averaged between 80 and 100 operations per second.

Each index method was tested by building an index on the first 5000 words in the system spelling dictionary. The fifteen character key is the word itself; the four character value is the word's index in the dictionary. 250 keys known to be in the index were retrieved in random order. The results of this test are presented in table 1.

For comparison, the same tests were performed using the indexed sequential (ISAM) file facility provided by the record management services component of VMS⁵. While VMS ISAM consumed less CPU time in constructing the index, it produced a file that was noticeably larger and more than half empty. The B-tree index method performed significantly better than VMS ISAM in the retrieval test in terms of both CPU time and disk reads. The lower page fault statistic for retrieval from the ISAM file is explained by the fact that the retrieval was done in the VMS command language rather than by an explicit program. The command language interpreter is permanently resident in memory, while a program must be paged in from disk as it runs.

The extendible hash index method was not evaluated in this test. The implementation is not as efficient as was hoped for; it produced a file of over 1000 pages for the test data. Extendible hashing works best with large hash buckets; i.e., on the order of 2^{13} bytes. In this implementation a bucket is a single page (2^{10} bytes), and the

directory is split many more times than it ought to be just in order to make more space in the index. The retrieval performance of the extendible hash index method is adequate, but the insertion performance and space utilization are unacceptable.

Conclusion

Aside from the poor performance of the extendible hash index method, several other problems require attention. Currently, the module in the IM/SM layer which allocates variable-length byte strings in a text store cannot handle strings longer than the length of a database page. This is not too serious a limitation, but some facility should be provided for storing large contiguous chunks of data.

Two minor enhancements would significantly improve the performance of the SEG layer. When new pages are created in a segment, free page positions are found by recursively traversing the map pages of the segment. Typical segments will have only a few map pages anyway, so the search is not overly expensive. However, several page accesses could be saved by using a bitmap to keep track of which pages in the segment are defined and which are undefined. This strategy would only cost 8 pages (of 2^{10} bytes each) for a completely full segment and would make the page allocation process much more efficient, especially in large segments.

The second enhancement is to provide some space in the segment header structure for use by the access methods. Each access method must store some information about how it structures data in a segment; typically this is done by reserving the zeroth page of the segment for that purpose. But pages are large and usually only ten or twenty bytes of data are involved. It would be more efficient to allow access methods to store and retrieve an *access method header* which is maintained in part of the segment header. This would reduce the number of direct mapping pointers that can be stored in a segment header, but the loss would be repaid by not having to access an additional page for necessary structural information.

At the moment, alphabetic case is significant in all the index methods which support character string keys. It is not clear whether ignoring case for character string keys in indices is a desirable thing or not; for many this is a religious issue. This problem should be thought out and a way to enable and disable case folding should be implemented.

The single pattern-matching index method, the associative string space, supports only a limited sort of pattern. Full regular-expression pattern matching¹⁸ would significantly extend the power of associative string indices. One problem lies in implementing a regular-expression search.

algorithm that would not significantly degrade the performance of the associative string index method.

Two new index methods are being considered. An order-preserving linear hash index method will eventually replace the doomed extendible hash index method. A novel index technique based upon linear interpolation has been invented by Burkhard³. This index method can perform efficient *orthogonal range queries*—retrievals based on target keys from several orthogonal key spaces.

The conflicts between the I/O activities of a database file system and the virtual memory paging activities of the operating system under which it runs have been thoroughly investigated¹². It is wasteful to have two paging mechanisms working in competition against each other. Many operating systems, including VMS, provide the ability to directly map files into memory; i.e., system calls are provided to tell the system to page a certain area of memory from a given file instead of from the system paging file. Initial tests have been performed using this technique and it looks very promising. The integration of this facility into the file system would replace the current PC and PIO layers and would require a slight redefinition of the interface functions provided by the PC layer.

Acknowledgments

Thanks to Richard Billington, Peter Buneman, Sharon Perl, and Ira Winston for their advice and encouragement throughout this project.

References

- [1] M. M. Ashtrahan et. al. System R: A Relational Approach To Database Management. *ACM Transactions on Database Systems* 1(2), June 1976.
- [2] R. Bayer and C. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1(3), 1972.
- [3] W. Burkhard. Interpolation-Based Index Maintenance. *BIT* 23:274-294, 1983.
- [4] C. J. Date. *An Introduction to Database Systems*, third edition. Addison-Wesley, 1981.
- [5] —. *VAX-11 Record Management Services Reference Manual*, version 3.0. Digital Equipment Corporation, Maynard, Massachusetts, 1982. DEC publication no. AA-D031D-TE.
- [6] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible Hashing—A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems* 4(3), September 1979.
- [7] J. Gray. Notes on Database Operating Systems. In R. Bayer, R. M. Graham, and G. Seegmuller (editors), *Lecture Notes in Computer Science*. Volume 60: *Operating Systems*. Springer-Verlag, Berlin, 1978. Appeared also in an IBM San Jose Research Laboratory report, 1977.
- [8] D. L. Kashtan. *EUNICE: A System for Porting UNIX Programs to VAX/VMS*. Technical Report, SRI Artificial Intelligence Center, October 1982.
- [9] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
- [10] D. E. Knuth. *The Art of Computer Programming*. Volume 3: *Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1973.
- [11] B. W. Lampson. Atomic Transactions. In B. W. Lampson, M. Paul, and H. J. Siegart (editors), *Lecture Notes in Computer Science*. Volume 105: *Distributed Systems—Architecture and Implementation*. Springer-Verlag, Berlin, 1981.
- [12] T. Lang, C. Wood, and E. B. Fernandez. Database Buffer Paging in Virtual Storage Systems. *ACM Transactions on Database Systems* 2(4), December 1977.
- [13] M. E. Lesk. *Some Applications of Inverted Indexes on the UNIX System*. Computing Science Technical Report 69, Bell Laboratories, June 1978.
- [14] W. H. Paxton. *A Client-Based Transaction System To Maintain Data Integrity*. Technical Report CSL-80-3, Xerox Palo Alto Research Center, March 1980.
- [15] M. B. Reinhold. *A Database Construction Kit*. Technical Report, Moore School, University of Pennsylvania, 1984. To appear.
- [16] D. W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems* 6(1), March 1981.
- [17] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM* 24(7), July 1981.
- [18] K. Thompson. A Regular Expression Algorithm. *Communications of the ACM* 11(6):419-422, 1968.

Structure-level Specification: Program Design

Margaret J. Davis and A. Toni Cohen

Department of Computer and Information Sciences
University of Delaware, Newark, Delaware 19716

Abstract

Formal languages for requirements definition and design specification of software systems can be classified by descriptive purpose. System-level languages model the externally-observable behavior of a software system in its environment. Structural-level languages describe a high-level design for the software system, and relate the design to the system model. Local-level languages specify the behavior of each software module. Structural-level specification is explained through examples drawn from a specification for an airlines reservation system.

Introduction

The idea of using formal techniques to record the specification and design for a software system has been a subject of research since the 1970's. One early success, PSL/PSA [14], used a formal language to analyze a model of a proposed system recorded in a database. Other efforts produced languages, such as AFFIRM [5], whose purpose was to describe what a particular software module was supposed to do rather than how it was to be done. These languages essentially formalized and refined the use of pseudocode as a design technique.

More recently, research has focused on creating wide-spectrum languages, such as GIST [2] and PAISLey [15, 16], which are suitable for requirements definition through low-level design. These languages permit a system designer to create an executable model of the proposed system. In effect, they apply the technique of prototyping to the design of software. Furthermore, it is expected that a prototype created with this technique can be translated into an efficient software system. Although it is assumed that human guidance [3] will be necessary, the changes are affected through successive applications of appropriate transformations,

in much the same manner as compilers change high-level languages into machine code.

Guttag [7, 8], however, proposes that the activities which comprise requirements definition and design should be supported by three distinct types of specification languages classified by their descriptive purpose. A system-level language would describe the application-oriented behavior to be implemented. A structural-level language would describe a high-level design and the interactions among its parts. A local-level language would describe the language-oriented behavior of each part of a high-level design.

One justification for viewing the specification process as composed of three phases is that this view mirrors the current wisdom on good programming methodology: to postpone decisions about lower level details until all decisions are made for higher levels. Decisions regarding the effect of an operation in a program unit are at a lower level of abstraction than decisions about how to functionally decompose the desired systemic behavior, which, in turn, are at a lower level than decisions regarding what behavior to include in the systems model.

Definition Of Each Level

System specification defines a conceptual model of the software's desired behavior. The modeled behavior is that which can be observed by persons who will use the software, and may include a description of environmental stimuli [1, 10]. It includes descriptions of entities, relationships between entities, and events. Entities are things which are distinguishable; they may be tangible or intangible. Events are changes in the model's state, where state means the current values of all model components. Events may create or destroy entities or may cause

* This research supported by System Development Foundation grant G-403.

changes in the values of an entity's properties. The model for an inventory control system would define entities such as parts, relationships such as sub-assemblies, and events such as stockouts. GIST [6], PAISLey, and TECTON [11] are suitable languages for system-level specification.

Structural specification organizes the behavior of the system model into data and procedural abstractions. That is, it defines data structures and a functional decomposition whose behavior implements that of the system model. The details given for an abstraction concern its i/o behavior (names of operations, names and types of i/o parameters), its interaction with other abstractions, and its relationships¹, if any, with components of the system-level specification.

Each abstraction is considered an independent program unit. A data abstraction is a data structure together with the operations that construct and access it. A procedural abstraction is a collection of procedures that share data. The use of an abstraction is analogous to the use of operating system calls to create and read or write to a disk file. The effects and interfaces of each call are known, but how the called procedures accomplish their tasks is not. Indeed, knowledge of such details is of no importance to a user, and may be freely changed in later versions of the operating system.

No languages have been designed as yet for structural-level specification. The examples, which appear in Figure 2, demonstrate semantic constructs which could be included in such a language. For instance, programming languages provide facilities for iteration and structuring (Fortran's DO-loop and subroutine respectively). One goal of this research is to suggest facilities that a structural-level specification language would provide.

Local specification defines the behavior of each program unit by describing the effect of each of its operations. The description of an operation's effect may be implicit (in the form of axioms) or explicit [4]. The description of each abstraction will also include definition of data structures and operations which are not accessible from outside the abstraction. A programmer who uses a language provided function

such as PASCAL's *new* does not need to know what other procedures *new* calls or what data it uses, in fact, the knowledge could be detrimental. Some of the languages which have been used for local-level specification are AFFIRM and LARCH [9].

This classification represents a hierarchy of specification ordered by level of abstraction, where the system specification is the most abstract and contains the least detail and the local specification is the least abstract and contains the most detail.

Informal Description

Remaining discussion of system and structural specification will refer to a single example problem - an airline reservation system. Although the system described is not realistic, the example demonstrates some of the complexity involved in putting formal specification into practice.

Informally, the problem is to provide an airline reservation system for a single airline with flights between New York, Chicago, Atlanta, and Los Angeles. A customer can request information about available flights between two locations on a particular date. The customer can reserve a seat for a particular flight on a specific date, and can request certain special treatments, such as seating in a non-smoking section. The airline can retrieve a passenger list for a particular flight and date, which will also list passengers' special requests. The information recorded about each flight includes information such as departure and arrival times.

System-level Description

The system level specification presented in Figure 1 is a model of the airline reservation system described above. It may be viewed as a definition of a database of the model's state information along with a description of all possible ways to alter its state. The language we use borrows freely from GIST [6]. Three types of components (entities, relationships, and events) are described.

Each entity description defines a domain. As can be seen in Figure 1, domains may be:

- enumerated as are PLANE, SECTION, and SPECIAL_CARE;
- composed from other domains as are FLIGHT_NO and DATE;
- or constraints on some larger domain as is SEATING.

¹The definition of structural specification given in [7, 8] does not include the relationships of abstractions to system-level behavior. However, knowledge of these relationships is invaluable to the process of software maintenance. It may also encourage the development and comparison of alternate designs.

System-level Specifications

** entity descriptions **

DEPART_LOC IS {NY, CHICAGO, ATLANTA, LA}
ARRIVE_LOC IS {NY, CHICAGO, ATLANTA, LA}
DEPART_TIME IS <HOUR, MIN>
ARRIVE_TIME IS <HOUR, MIN>
FULL IS *BOOLEAN*
NUM_SEATS_AVAIL IS *NATURALNUM*
GATE IS *ALPHACHAR*
SURNAME, FIRSTNAME IS *ALPHACHARSEQUENCE*
PASSENGER IS <SURNAME, FIRSTNAME>
SECTION IS {NON-SMOKING, SMOKING}
FOOD IS {REG, KOSHER, LO-CAL, VEGETARIAN}
SPECIAL_CARE IS {NONE, CHILD, MEDICAL}
FLIGHT_NO IS <*ALPHACHAR*, *NATURALNUM*>
DATE IS <MONTH, DAY, YEAR>
PLANE_MODEL IS {L1011, DC9, 747, 727}
SEATING IS $50 \leq \text{NATURALNUM} < 224$

** relationship description **

FLIGHT (DEPART_LOC, ARRIVE_LOC, DATE,
DEPART_TIME, ARRIVE_TIME,
FLIGHT_NO, NUM_SEATS_AVAIL,
GATE, FULL, PASSENGER_LIST)

PASSENGER_LIST (PASSENGER, FLIGHT_NO,
DATE, SECTION,
FOOD, SPECIAL_CARE)

PLANE_CAPACITY (PLANE, SEATING)

TICKET (PASSENGER, FLIGHT_NO, DATE,
GATE, DEPART_TIME)

** event descriptions **

RESERVE_SEAT (FLIGHT_NO, PASSENGER,
SECTION, FOOD,
SPECIAL_CARE, DATE)
INSERT PASSENGER_LIST
MODIFY FLIGHT
CREATE TICKET

QUERY_FLIGHT (ARRIVE_LOC, DEPART_LOC,
DATE)
SELECT FLIGHT

LIST_PASSENGERS (FLIGHT_NO, DATE,
DEPART_TIME)
SELECT FLIGHT

DEMON CHANGE_FULL
FLIGHT:NUM_SEATS_AVAIL == 0;
FLIGHT:FULL == *TRUE*

Figure 1

The names given entities would normally use the vocabulary of the client. It is expected that experts in system level specification will be constructing and using the model [2, 8], so the domain descriptions do not have to be understood by the clients. However, researchers working on GIST have developed a tool which translates the GIST specification into natural language [12, 13].

Each relationship description names the relationship and then lists the entities related. FLIGHT, PASSENGER_LIST, PLANE_CAPACITY, and TICKET are relations defined in the example. Notice that TICKET refers to a physical object, which exists in the environment of the airline reservation system; while PLANE_CAPACITY refers to a property of a physical object.

Each event description has a name, lists what entities are used as input, and lists subevents which constitute the effect of the named event. Subevents may be operations (insert, delete, create, modify) found in database manipulation languages or may refer to other events defined for the model. Specific sequencing of the subevents should be explicitly indicated only when required [1]. A DEMON causes a model to change state whenever its conditional expression becomes true. In Figure 1, the definition for demon CHANGE_FULL means that if the number of available seats for a flight is reduced to zero, that is, the flight is full, then the attribute FULL for that flight takes on a value of true.

Executing this model to validate its behavior would require first describing an initial configuration of its entity instances and relationships and then simulating its events.

Structural Description

For clarity, certain conventions have been used in the examples of structural-level specification displayed in Figure 2. Identifiers from the system-level specification appear in uppercase; those from the structural-level in lower-case. Parameter lists are enclosed by parentheses () in the PASCAL style of (IDENTIFIER:TYPENAME). Input/output mappings,

([TYPE × TYPE] → TYPE)

which list the types of input and output parameters, are also contained within parentheses. The input types are enclosed in square brackets [], with each type name separated by an ×; the output types follow a →. Language keywords appear in **boldface**. In the text, structural-level identifiers appear in lower-case *italics*.

Data Abstractions

A data abstraction which represents a system-level entity or relationship inherits a value set and behavior from that entity or relationship. The keyword **realizes** indicates this correspondence. In Figure 2, *loc* is a data abstraction whose domain is constructed from the legal values for DEPART_LOC and ARRIVE_LOC. A more complex form of **realizes** appears in the description of *psngrname*. The abstraction *psngrname* represents the system-level component PASSENGER, which, in turn, is a composite of the two system-level components SURNAME and FIRSTNAME. The use of **as** makes explicit which identifier represents which system-level components.

The phrase

restricts IDENTIFIER to EXPRESSION

constrains the value set of an abstraction. The value set for *loc* is constrained to be exactly {NY, CHICAGO, ATLANTA, LA}. The value set definition for *loc* is not meant to imply that its only valid implementation would use an ASCII representation of its members, but rather that the implementation would only recognize representations for NY, CHICAGO, ATLANTA and LA as valid occurrences of the parameter *l*. Sometimes it is more convenient to specify the constraint on a parameter of an abstraction as the clause restricting *c* to [A..Z, a..z] in the definition for *names* indicates.

The keyword **is-a** means the identifier to its left has similar properties to the identifier on its right, but not necessarily the same value set. Thus, *loc* is an abstraction which has the same properties as the abstraction *set*. The definition of *set* includes a parameter, thus *loc* specifies a parameter *l* with type *charseq*. Since no operations are listed for *loc* it is assumed that its operations are identical in name and effect to those of *set*.

Each data abstraction has a set of operations besides a set of values. The phrase **has operations** introduces definitions for these operations. Normally, the operations for an abstraction defined using an **is-a** are copied from the abstraction named to the right of that keyword, with appropriate substitutions of identifier names and types. The phrase **adds operations** indicates that operation definitions following it are to be added to those which have been copied. The operation set for *capacity* has all the operations of *posint* plus the two additional operations named *whatismax* and *whatismin*. If changes are desired to the original definitions, then the operations are defined again with a **has**

Structural-level Specification

<pre> ** Data Abstractions ** loc is-a set(l:charseq) realizes DEPART LOC, ARRIVE LOC restricts loc to {NY, CHICAGO, ATLANTA, LA} avail is-a posint realizes NUM_SEATS_AVAIL derives maxcapacity from capacity.whatismax restricts avail $\leq$ maxcapacity names is-a sequence(c:character) realizes SURNAME, FIRSTNAME psngrname is-a pair(sn:names,fn:names) realizes PASSENGER as psgnrname, SURNAME as sn, FIRSTNAME as fn schedule is-a file(flt:flight_info) realizes FLIGHT has operations create($\rightarrow$schedule) add_flight([schedule $\times$ fnum $\times$ loc $\times$ loc $\times$ times $\times$ times $\times$ dates $\times$ planes $\times$ gates] $\rightarrow$schedule) delete_flight([schedule $\times$ fnum]$\rightarrow$schedule) findbyloc([schedule $\times$ loc $\times$ loc $\times$ dates] $\rightarrow$flight_info) findbynum ([schedule $\times$ fnum $\times$ dates] $\rightarrow$flight_info) </pre>	<pre> psngrsum is-a table(elm:valuepair) uses psindex for index, pskeys for keynames posint is-a integer restricts posint > 0 set(elm) has operations member-of([set $\times$ valuepair] $\rightarrow$boolean) charseq is-a sequence(elm) uses uppercase for elm file(elm) uses record for elm has operations create($\rightarrow$file) insert([file $\times$ record] $\rightarrow$file) search([file $\times$ valuepair] $\rightarrow$record) delete([file $\times$ record] $\rightarrow$file) table(elm) requires keynames and index has operations insert([table $\times$ valuelist $\times$ index $\times$ keynames]$\rightarrow$table) print([table $\times$ index] $\rightarrow$values) </pre>
---	---

Figure 2

operations construct. The specification for *schedule* has been defined this way because the input/output mappings are different from those of *file*, although the result of the operations is the same.

The keyword **requires** indicates when an abstraction needs auxiliary type definitions. The abstraction *table* requires two types - *keynames* and *index*. If *table* appears to the right of the keyword **is-a** then the abstraction being defined must include a construct beginning with **uses** to indicate what abstractions are to be used for *keynames* and *index*. Further, those abstractions must be defined via **is-a** *keynames* and *index* respectively.

An abstraction may use values from another. The phrase

derives IDENTIFIER from OPERATIONSEQUENCE indicates that the identifier's value is obtained from that operationsequence. An operationsequence is a composition of operation names and argument lists. In the example specification for *avail*, *maxcapacity* is derived from the operation *whatismax* defined for abstraction *capacity* which has no arguments.

Procedural Abstractions

The specification of a procedural abstraction is very similar to that of a data abstraction. Where a data abstraction may have a parameter list, the procedural abstraction has an input/output mapping. Correspondences with system-level events are defined with **realizes**; local data is defined with **uses**; use of another abstraction as a template is defined with **is-a**; operations are defined with **has operations** or **adds operations**; and, allowable values of local data are constrained with **restricts**.

Discussion

Most of the benefits derived from using formal specifications are those which can be achieved with a disciplined design methodology. However, a formal specification has the advantage that it can be computer-analyzed. Thus, formal specification languages are designed as language systems, that is, a language and a set of tools to manage and process what has been written in the language [8, 1, 2].

Given both a system and structural-level specification, some possibilities for analyses and assistance that such a language system could provide are:

- verify that each system-level component has a structural-level realization;

- verify that a data abstraction that realizes an entity has the correct value set;
- verify that an event, a relationship, or a composite entity realized by an abstraction has realizations for each of its parts, and that these parts are correctly typed;
- verify agreement of an invocation of operation with its input/output mapping;
- report what structural-level abstractions are affected by a change made in the system-level specification; and,
- report what abstractions are used by another and classify which usages are only affected by a change in an input/output mapping (decoupled).

All of the above relieve the designer of tedious work. The first four are analogous to static checks performed by a compiler; the last two are analogous to graphical representations (form of flowcharts) of interactions between components. A more interesting analysis would check that the system-level behavior is represented at the structural-level. For instance, *loc* was defined as identical in behavior to *set*. If it could be verified that *set* has the essential properties of an enumerated domain, that would be useful. It would be interesting to develop some measure of a design's flexibility or modularity so that two specifications could be compared.

Acknowledgements

Thanks go to D. Chester, T. Myers, and R. Weischedel for encouragements, questions, and criticisms.

References

1. Balzer, R., Goldman, N. "Principles of Good Software Specification and their Implications for Specification Language." *IEEE Trans. S.E.* (Jan 1979), 58-67.
2. Balzer, R., Goldman, N., Wile, D. Operational Specification as the Basis for Rapid Prototyping. USC/ISI, Oct, 1981.
3. Balzer, R. "Transformational Implementation: An Example." *IEEE Trans. S.E.* (Jan 1981), 3-14.
4. Claybrook, B. "A Specification Method for Specifying Data and Procedural Abstractions." *IEEE Trans. S.E.* (Sep 1982), 449-459.

5. Gerhart et al. An Overview of AFFIRM: A Specification and Verification System. INFORMATION PROCESSING 80, 1980, pp. 343-347.
6. Goldman, N., Wile, D. GIST Language Description - Draft. USC/ISI, Sep, 1980.
7. Guttag, J. A Few Remarks on Putting Formal Specification to Productive Use. In *Program Specification*, Springer-Verlag, 1982, pp. 370-380.
8. Guttag, J., Horning, J., Wing, J. Some notes on Putting Formal Specification to Productive Use. Tech. Rept. CSL-82-3, XEROX PARC, Jun, 1982.
9. Guttag, J., Horning, J. Preliminary Report on The Larch Shared Language. Tech. Rept. CSL-83-6, XEROX PARC, Oct, 1983.
10. Heitmeyer, C., McLean, J. "Abstract Requirements Specification: A New Approach and Its Application." *IEEE Trans. S.E.* (Sep 1983), 580-589.
11. Kapur, D., Musser, D., Stepanov, A. TECTON: A Language for Manipulating Generic Objects. In *Program Specification*, Springer-Verlag, 1982, pp. 402-414.
12. Swartout, B. GIST English Generator. USC/ISI, Apr, 1982.
13. Swartout, B. GIST Behavior Explainer. USC/ISI, Mar, 1983.
14. Teichroew, D., Hershey, E. "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems." *IEEE Trans. S.E.* (Jan 1977).
15. Zave, P. "Operational Approach to Requirements Specification for Embedded Systems." *IEEE Trans. S.E.* (May 1982), 250-269.
16. Zave, P. Operational Specification Languages. ACM '83, ACM, Oct, 1983.

Odradek - A Prolog-Based Lisp Translator

Herb Jellinek
Research and Development
SDC — A Burroughs Company
P.O. Box 517
Paoli, PA 19301

Abstract

Odradek is a Prolog program that implements a Lisp-to-Lisp source code translator. It is based on a set of rules stating equivalences between forms in the two Lisp dialects. Because Odradek is implemented in "pure" Prolog, it is bidirectional. Odradek has been used to translate a number of large and small Lisp programs from one dialect to another.

Introduction

As the commercial and academic use of the programming language Lisp increases, more attention is being paid to the problem of incompatible Lisp dialects. We at SDC-LBS, for instance, are almost daily confronted with this problem. We use two major Lisp systems in our work: Franz Lisp[2] on our VAX-11/780¹, and Interlisp-D[3] on Xerox Lisp Machines². At times it seems that the only similarity between the two dialects is their mutual use of S-expressions to represent programs and data. As we have a large investment in Franz Lisp code that we wish to run under Interlisp-D, we needed to develop the capability of rapidly translating between the two.

Our first efforts employed the Interlisp **Transor** subsystem[4]. **Transor** lets the user enter a series of *editor scripts* - programs written in the language of the Interlisp symbolic editor, and then uses these scripts to perform the translation. It soon became obvious that **Transor** was unusable for our purposes: the edit scripts are difficult to write or read; translation is unidirectional; the **Transor** documentation is quite old and somewhat out-of-date.

We realized that we had the makings of a very good translator in our Prolog system, one of the very programs we sought to translate! In

fact, after this insight, we had a skeleton translator up and running within 30 minutes. This paper, then, is an explanation of the design of our Prolog-based Lisp translator, named **Odradek**³.

The translator is based on a database of rules describing equivalence between expressions in both dialects. It repeatedly reads S-expressions from a file, matching each S-expression to be translated against the appropriate grammar rule, which rewrites the S-expression into a form suitable for the target Lisp dialect. This process is repeated recursively on the non-terminal symbols in the source S-expression until the translation is completed. The Prolog implementation of this algorithm is unusually straightforward and transparent; it also has the advantage of being invertible: it can translate in either direction with no changes to the code. To see how this is possible, we must first examine Prolog, for it is from Prolog that we gain most of Odradek's power.

Prolog

Prolog ("PROgramming in LOGic") is a fairly young language which is gaining increasing acceptance in artificial intelligence research. It features powerful pattern-matching (unification), search, and backtracking capabilities, as well as very flexible data structures. It is handy for rapid prototyping of complex systems and has well-defined semantics. Prolog programs are theorems in a Horn-clause based predicate calculus; Prolog runs a program by attempting to prove (actually, disprove) a given query, using Robinson's resolution theorem proving technique.[1] Prolog clauses can be read both declaratively and procedurally.

The simplest Prolog program consists of a single *clause*, asserting a single fact; for instance,

¹ VAX is a trademark of Digital Equipment Corporation.

² Xerox is a trademark of Xerox Corporation.

³ The name is from Franz Kafka, quoted in *The Book of Imaginary Beings* by Jorge Luis Borges, pp. 171-173, E. P. Dutton, 1978.

(likes Floyd espresso).⁴

states that "Floyd likes espresso." This particular statement is called a *ground clause*, because it contains no variables. We can also create *rules*, which are predications, as in the fourth clause of the following:

```
(is-warm tea).
(is-warm espresso).
(is-good espresso).
(likes Floyd x) if (is-warm x) (is-good x).
```

The first three ground clauses state some obvious facts. The fourth clause is more complex, consisting of a *head*, or consequent ("(likes Floyd x)") and a *body*, or antecedent ("(is-warm x) [and] (is-good x)"). (There is an implicit conjunction between the two terms in the body.) The symbol "x" is a variable. Thus, the declarative reading of this rule says that "Floyd likes *x* if *x* is warm and *x* is good." The procedural reading is: "To prove that Floyd likes *x*, prove that *x* is warm and that *x* is good." If we query Prolog with

```
(likes Floyd what)?
```

Prolog will reply

```
((likes Floyd espresso))
```

binding the variable what to the value "espresso." Querying

```
(likes person thing)?
```

will bind person to "Floyd" and thing to "espresso." When given the query (likes person thing), Prolog will search its database, attempting to **instantiate** (bind) variables in such a way that the query is satisfied. If Prolog fails in an attempt to **unify** two data elements, it backtracks, exploring all alternatives in succession until either the unification succeeds or all possibilities are exhausted. (The latter case is called **failure**). In the above example, (is-warm x) first matched (is-warm tea), binding x to tea, but (is-good tea) failed to unify with any clause in the database. Thus, a failure was caused, and x was unbound. Prolog generated the next alternative, binding x to espresso, which succeeded. As you can see, unification is a powerful technique; variables can play the role either of input or output, depending on whether they occur bound or unbound in the execution of a particular program. Odradek relies on this ability to achieve its bidirectional translation.

Simple Translation Rules

With these ideas in mind, we will now explore the implementation of Odradek. The Odradek translator consists of two pieces: the first, a database of "translation rules" (clauses) which declare the equivalence between two Lisp expressions, and the second, a read-translate-print loop which is Odradek's driver. Odradek attempts to match pieces of code against the rules in its database; any variables (names that begin with "") present in the rules will have appropriate values bound to them upon return.

For instance:

```
(ltrans PossibleAtomOrConstant PossibleAtomOrConstant) if
  (atom PossibleAtomOrConstant).
```

is the identity transformation, stating that Lisp atoms and constants are their own translations;

```
(ltrans (* Factor1 Factor2)
  (ltrans NewFactor1 NewFactor2)) if
  (ltrans Factor1 NewFactor1)
  (ltrans Factor2 NewFactor2). (1)
```

is the clause that states that the "*" (product) of two arguments in Franz Lisp translates to "ITIMES" of two arguments in Interlisp, if the arguments are also translatable. We can present goals (queries) to Prolog, which it will attempt to match against (unify with) the heads (portion before the "if") of all clauses. If the match is successful, the body (portion after the "if") of the matching clause becomes the new collection of goals to match against, and so on, recursively, until a match fails. Thus, if we present the following goal to Prolog,

```
(ltrans weeklyPay Interlisp)
```

the variable Interlisp will become bound to weeklyPay, and if we query

```
(ltrans (* weeklyPay 52) Interlisp)
```

it will succeed with the variable Interlisp bound to (ITIMES weeklyPay 52), having matched against rule (1) given above. Querying

```
(ltrans FranzLisp
  (ITIMES (ITIMES weeklyPay 52) 6))
```

succeeds with FranzLisp bound to (* (* weeklyPay 52) 6).

More Complex Rules

Some transformations are not quite so direct. For example, *fexprs*, functions that receive their arguments unevaluated, must receive special care in translation. An example of such a function is **cond**, which considers its indefinite number of "arguments" to represent a nested if-then-elseif conditional. Without special knowledge of **cond**'s structure, the translator would believe each argument to be a

⁴ SDC's Prolog system, called HCPProlog, uses a non-standard syntax that is similar to that of Lisp. HCPProlog is described in detail in Karl Puder's paper in this volume.[5] Its predecessor, HCPRVR, is described in [6].

function-call, as they would be if **cond** were a **lambda** function, which most Lisp functions are.

The following conditional expression might cause a more naive translator to generate an incorrect translation:

```
(cond ((null car) (print-it "You have no car!"))
      (car (print-it "Driving car " car)))
```

In this form the value of the *variable* **car** is referenced; nowhere is the *function* **car** invoked. If the *function* **car** in Lisp1 is declared to be equivalent to the function **FIRST** in Lisp2, the result of naive translation would be

```
(COND ((null car) (print-it "You have no car!"))
      (FIRST (print-it "Driving car " car)))
```

which is certainly not what was intended. To resolve such problems auxiliary relations are used in the translation. In this case, we use one that views the **cond**-arms correctly as a list of if-thens:

```
(ltrans (cond . _CondArms) (COND . _NewCondArms)) if
(ltransCondArms _CondArms _NewCondArms). (2)
```

```
(ltransCondArms ((_Condition . _Consequents) . _Rest)
  ((_NewCondition . _NewConsequents) . _NewRest)) if
(ltrans _Condition _NewCondition)
(ltransTail _Consequents _NewConsequents)
(ltransCondArms _Rest _NewRest). (3)
```

```
(ltransCondArms () ()). (4)
```

```
(ltransTail (_Head . _Tail) (_NewHead . _NewTail)) if
(ltrans _Head _NewHead)
(ltransTail _Tail _NewTail). (5)
```

```
(ltransTail () ()). (6)
```

Clause (2) states that in order to translate a **cond** form from one dialect to the other, we must first translate the **cond**'s arms, or conditions. This translation is done using the auxiliary relation *ltransCondArms*. The head of *ltransCondArms* is designed to unify with a list of canonical **cond**-arms. Such **cond**-arms consist of a single condition followed by an arbitrary number of *consequents* (i.e. Lisp function-calls.) The action of *ltransCondArms* is to translate the condition, translate the *consequents* (using the auxiliary *ltransTail* relation), and then to recur on the remaining **cond**-arms. Clause (4) serves to terminate the recursion, and clauses (4) and (5) define *ltransTail*, which maps *ltrans* across a list of forms.

Bidirectionality

Another *ferpr* that we need to translate is the Franz **do** (loop) function. (Odradek's rules for translating **do** are somewhat lengthy, and

we will not present them here.) Querying

```
(ltrans (do loop-var 0 (1+ loop-var)
          (= loop-var 10)
          (print loop-var) (terpr))
  _Interlisp)
```

binds *_Interlisp* to

```
((LAMBDA (loop-var)
  (while (NOT (EQP loop-var 10))
    do
      (PROGN (print loop-var) (TERPRI))
      (SETQ loop-var (ADD1 loop-var))))
  0)
```

As an example of Odradek's bidirectional translation capabilities, executing

```
(ltrans _FranzLisp ((LAMBDA (loop-var)
  (while (NOT (EQP loop-var 10))
    do
      (PROGN (print loop-var) (TERPRI))
      (SETQ loop-var (ADD1 loop-var))))
  0))
```

binds *_FranzLisp* to

```
(do loop-var
  0
  (1+ loop-var)
  (= loop-var 10)
  (print loop-var)
  (terpr))
```

Current Status

At the present time, Odradek has 123 translation rules in its database. This is sufficient for translating most Franz Lisp forms into Interlisp, and many Interlisp forms into Franz

Lisp. (Interlisp has many more built-in functions and special forms than does Franz). It does not necessarily generate the most efficient translated code, but most of the inefficiencies are of a type that a good Lisp compiler can eliminate. An alternative to relying on the compiler is to add a second pass to Odradek that can recognize and simplify inefficient constructs.⁵ Rules have been added to Odradek that enable it to fix one common inefficiency/bug in Lisp code, namely the use of the *quote* function where the *function* function was called for.

Conclusion

We have described the design of Odradek, a bidirectional Lisp-to-Lisp translator written in a dialect of Prolog. Odradek has been used to translate several programs from Franz Lisp into Interlisp-D, including one program (the Prolog interpreter itself) that is over 2100 lines long. Our experience with Odradek has demonstrated two major points about Prolog: first, that it is a good prototyping language, allowing

⁵ Thanks to Prof. Tim Fain of the University of Pennsylvania for this suggestion.

the programmer to quickly generate running, useful code; and second, that it is a good language for writing production-quality code.

Acknowledgements

I'd like to express my thanks to my employer, System Development Corporation, for supporting my work; to my colleagues in the SDC Logic-Based Systems Group, for their friendship and many valuable suggestions; and to Tim Finin, for his critique of an early version of Odradek.

REFERENCES

- [1] J. Alan Robinson, *Logic: Form and Function*, North-Holland, 1979.
- [2] John Foderaro, *Franz Lisp Manual*, University of California at Berkeley, 1981.
- [3] Michael Sannella, *Interlisp Reference Manual*, Xerox Palo Alto Research Center, October 1983.
- [4] *ibid.*, Chapter 23.
- [5] Karl O. Puder, "HCPProlog: Yet Another Implementation of Prolog in Lisp," *Micro-Delcon '84 Proceedings*, IEEE 1984.
- [6] Daniel Chester, "HCPRVR: An Interpreter for Logic Programs," *Proceedings of the First Annual National Conference on Artificial Intelligence*, AAAI, 1980.

RULE-BASED COMPUTER ALARM ANALYSIS IN CHEMICAL PROCESS PLANTS

Daniel Chester and David Lamb
Department of Computer and Information Sciences

Prasad Dhurjati
Department of Chemical Engineering

University of Delaware
Newark, Delaware 19716

Abstract

The growing complexity of chemical process plants increases the risk of human misinterpretation of plant alarms when components fail. A computer program for analyzing alarm signals and other plant data is presented; this program, called Falcon, identifies probable causes of process disturbances with the aid of the artificial intelligence technology known as 'expert systems'. Two approaches are under development. In one, rules are applied to observed data to derive the likely causes of the disturbances. In the other, rules are combined to model the process components and their interconnections, in which observed effects are traced to their possible causes.

Introduction

As chemical process plants get more complex, they become more difficult to operate safely and efficiently. There will be more failures of components in a complex plant, resulting in disasters, forced shutdowns, or at least higher operating costs from sub-optimal plant operation. Plants are built with many sensors to measure the values of plant variables and report them to plant operators, and with many alarms to alert them to variables that deviate from their normal values. Increased plant complexity also means more data for the operators to attend to and increased chances for human error in the interpretation of that data.

Computers are now used to monitor the data available from sensors and to display it in a more digestible form. Their ability to make decisions suggests that they may be able to analyze alarm and sensor signals automatically and may be able to advise the operators about what is happening in the plant. One approach to alarm analysis is based on pattern recognition; a pattern of observable data is associated with each foreseeable

fault and compared with the observed data. The fault with the best matching pattern is the most likely explanation for the plant condition, provided only one fault is present. Berenblut and Whitehouse [1] use decision tables to do the required pattern matching. More discussion of this approach can be found in Himmelblau [2].

Another approach to alarm analysis is based on graph theory. The interactions of plant variables are reduced to a labeled directed graph representing cause-effect relationships. The graph is examined to find subgraphs that explain the observed data in terms of a single disturbed variable. This is the approach of Umeda *et al* [3] and Iri *et al* [4]. Andow [5] and Andow and Lees [6] use graphs in a similar manner.

Both the pattern matching and graph theoretic approaches are extensively reviewed in O'Shima [7], Lees [8] and in the collection of papers edited by Rasmussen and Rouse [9].

Other approaches are based on the symbolic processing technology of the computer science field known as artificial intelligence, specifically, the technology known as expert systems. In these, chains of qualitative reasoning are explored to deduce the fault causing an exception condition, either directly from the observed sensor data or with the help of a causal model of the monitored plant. Examples of direct deduction are given by Nelson [10] and Fox *et al* [11]. Underwood [12] and Yamada and Motoda [13] discuss examples using causal models. Our own work now applies expert system technology to the diagnostic task of identifying the probable causes of disturbances in chemical process plants.

Overview of Falcon

Organization

An expert can predict the effects of a hypothetical fault in a plant by deducing how disturbances produced by the fault would propagate through the components in the process. This requires knowledge of the immediate effects induced by a fault in a given component, and knowledge of how disturbances in the inputs of a normal component

This work was supported in part by a grant from E. I. DuPont De Nemours & Co., and The Foxboro Company.

will propagate to disturbances in the outputs. Some of this information can be derived from a mathematical model of the system, but some of it is best supplied directly by the expert. Once this knowledge is explicitly represented, a computer program can reason similarly to predict the observable effects of a given fault.

When a process component fails, plant operators see only the observed effects; the diagnostic problem is to find the fault or faults that caused them. Our program, called Falcon (Fault Analysis CONSULTant), reasons backwards from observed effects to possible causes by applying rules in various ways. Figure 1 shows that Falcon consists of five modules. These are the supervisor, simulator, monitor, fault analyzer and human/machine interface modules. When Falcon is field tested in a chemical process plant, it will monitor process variables directly; in the meantime the simulator module provides the real-time data that drives the rest of the system. All modules can run independently or under control of the supervisor.

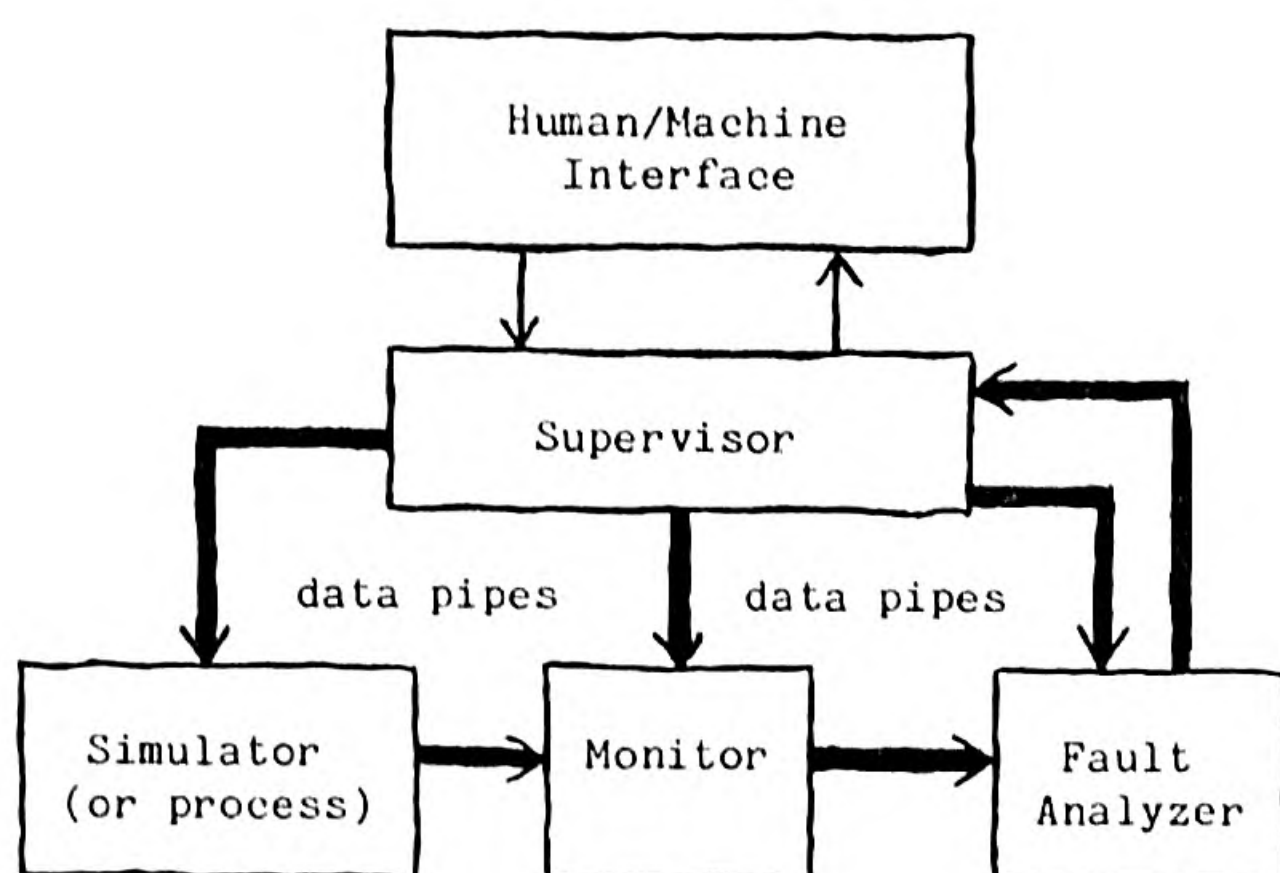


Figure 1. Modular organization of Falcon.

Supervisor

The supervisor allows the user to control several concurrently running tasks. Each of the other modules corresponds to a task; the supervisor provides an interface between these units, permitting a high degree of independence. Tasks communicate through 'pipes', which are data buffers that may be filled by one task and drained by another. (see figure 1.) The supervisor establishes pipes for both data transfer and control, and initializes each of the 'child' tasks. To approximate real-time phenomena, the simulator is allowed to send data only at fixed intervals, and the monitor is allowed to read only the most recently produced data vector.

Written in C, the supervisor provides the necessary read and write routines that permit the other modules to communicate through the pipes. The simulator is written in Fortran, and the monitor and fault analyzer are written in Franz Lisp. These modules have to run at the same time and must continue their primary functions when new input is not available.

Simulator

While Falcon is under development, the simulator module replaces the chemical process to be monitored. The simulator was built from DELSIM, a continuous system simulation software system modified to allow interprocess communication, to maintain a fixed ratio of simulation time to real time, and to operate under supervision by the supervisor. It contains a dynamic mathematical model of a chemical process, and can simulate a variety of faults in process components. When supervised, it accepts control vectors from the supervisor which may indicate a termination notice, time change, parameter change, or I/O status change. As the simulator executes the model, it generates vectors of observable variables at regular time intervals, which are placed in a pipe for the monitor. When run interactively (unsupervised), it allows a user to start simulation runs and to induce process component faults at will. In either mode, observable variables and some non-observable variables can be used to present pictorial representations of the system on an AED512 color graphics display terminal.

Monitor

The monitor is the module that continuously examines process data and determines whether there is a disturbance. It converts data received from the chemical process (either from plant instrumentation or from the simulator) to the proper structure and sends it to the fault analyzer. It also keeps the data in a data base, or history file, for further reference by the analyzer or the plant operators.

The data received from the process consists of numerical values representing the readings from various gauges or the status of alarms and switches. These are compared to the normal ranges for these process variables and converted to qualitative values, such as 'hi', meaning higher than normal, 'ok', meaning normal, and 'lo', meaning lower than normal. Trends and rates of change are computed from the data and converted also. When a disturbance (abnormal value) is discovered, the monitor sends the numerical values and their qualitative equivalents to the fault analyzer for analysis. The monitor notifies the analyzer whenever there is a significant change in the condition of the monitored process. If the analyzer or plant operators require information about past data, the monitor provides it while at the same time continuing to collect and process new data.

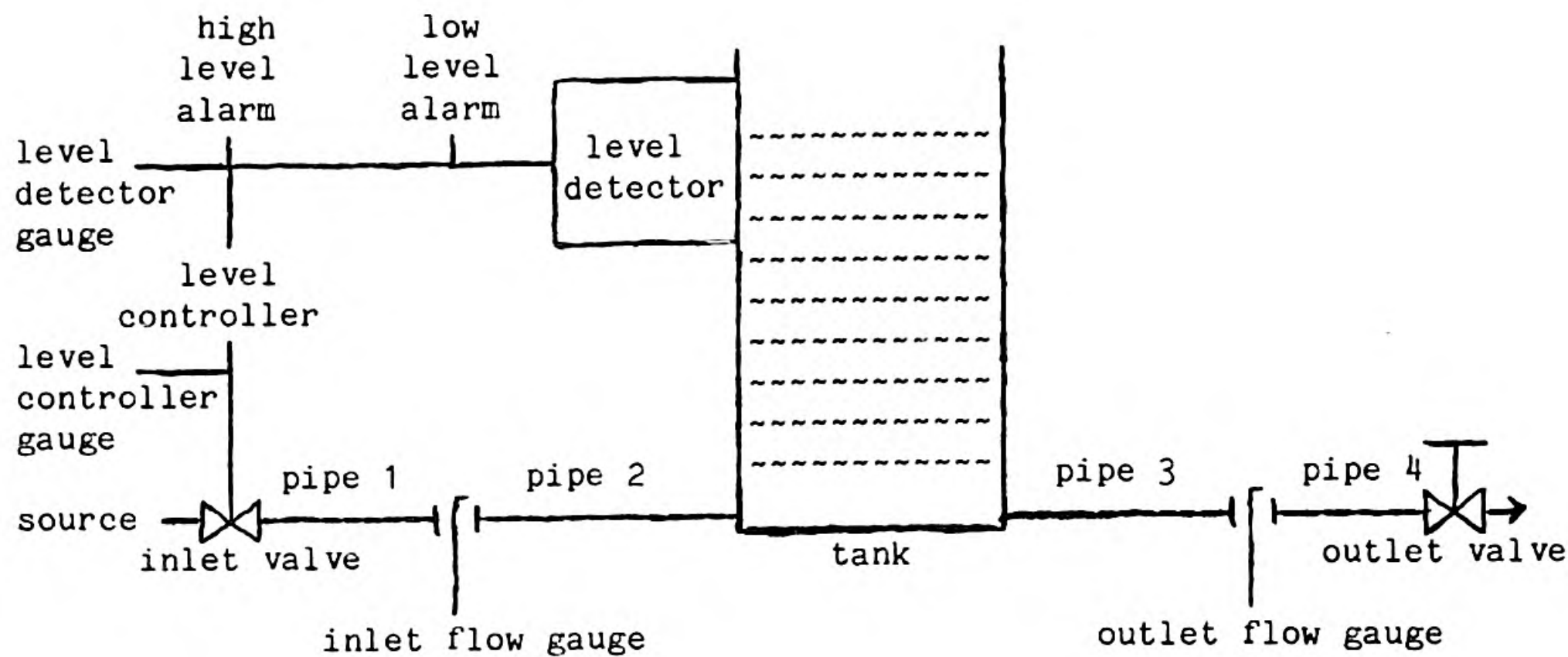


Figure 2. Simple buffer tank process.

Alarm Analysis

Example

The two approaches to alarm analysis that Falcon can take will be illustrated by the simple example of a process shown in Figure 2. This is a buffer tank designed to maintain a constant pressure and constant flow in its outlet. Water comes from a high pressure source producing a flow into the inlet valve. From there the water flows into a tank and then out through the manually controlled outlet valve. The level controller regulates inlet valve continuously to maintain the water level in the tank. Plant operators can observe the signals (or readings) from the high level alarm, the low level alarm, the level detector gauge, the level controller gauge, and the two flow gauges.

Suppose that the buffer tank process is in a condition that can be described qualitatively by the readings in Figure 3. Both fault analysis modules conclude that the inlet valve may be broken in an open position, but they arrive at this conclusion in different ways, and explain their answers differently. The following two subsections explain the two fault analysis modules in more detail.

high level alarm signal is observed to be on
 low level alarm signal is observed to be off
 level detector gauge reading is observed to be hi
 level controller gauge reading is observed to be lo
 inlet flow gauge reading is observed to be hi
 outlet flow gauge reading is observed to be hi

Figure 3. Sample state of buffer tank process.

Fault Analyzer

There are two versions of the fault analyzer, corresponding to two different approaches to building expert systems. The first version reasons directly from the observed data, much as an experienced operator might draw conclusions by virtue of his prior experience with the process. The second examines a model of the process to find possible causes of a disturbance. This can be likened to an engineer who is consulted in an emergency; he examines schematics of the particular process to understand it fully and applies his general knowledge about chemistry, physics and plant equipment to locate elusive faults.

Human/Machine Interface

This module is the least developed part of Falcon. It is intended to provide plant operators with a communication channel to the rest of the system. Falcon will inform the operators of faults developing in the monitored process and answer their queries about its analysis. Present user interfaces to modules communicate with text strings; only the simulator has a graphics display. The human/machine interface module will use more graphics to make it easier for operators to comprehend all the data that Falcon presents to them. There are many issues that have yet to be explored before design of this module is completed; What information should be displayed graphically? What should be text? What will operators want to know when Falcon suggests a cause for a process disturbance? The interface will also be used to inspect the simulator module to simulate specified process component faults while it is running in place of the actual process.

Deductive Approach

Diagnosis. The fault analysis module using the deductive approach draws conclusions about various process variables and components with rules stored in a knowledge base. The conclusions in turn lead to more conclusions until all possible deductions have been made. All conclusions about possible faults are displayed and numbered for easy reference. The conclusions are stored internally as statements in a data base against which patterns are matched to decide when rules apply. This module produces the message in Figure 4 when it processes the data summarized in Figure 3.

```
1  fault is inlet valve stuck open
```

Figure 4. Diagnosis using deductive approach.

When plant operators ask for an explanation of the module's diagnosis, they get the response shown in Figure 5. The reasons for reaching a given conclusion are listed and numbered so that they in turn can be explained if desired.

```
> why 1
because
  2  condition of level controller is good
  3  gauge for level controller reads low
  4  actual height of water in tank is high
  5  gauge for inlet flow reads high

> why 2
because
  6  gauge for level detector reads high
  7  gauge for level controller reads low
  and high is opposite of low

> why 4
because
  8  gauge for level detector reads high
  9  gauge for outlet flow reads high
```

Figure 5. Explanation of diagnosis in Figure 4.

Knowledge Base. All the knowledge available to this module about the buffer tank process is stored in the knowledge base as rules. Each rule has four parts, two of which are tests for applicability of the rule, and two of which actions to be taken. The first part of the rule is a list of patterns which must be matched against statements in the data base. These patterns contain

variables that must be bound to parts of the statements that they match, and that must be bound to the same values if they appear in more than one pattern. When statements are found to match all the patterns, the lisp code in the second part of the rule is executed to test the bound variables in some arbitrary way. If the test succeeds, the third part of the rule can be executed; it is another piece of lisp code which carries out some arbitrary action, usually printing something on the terminal. The fourth part of the rule consists of a set of patterns for statements which are to be added to the data base as new conclusions.

The rule shown in Figure 6, rephrased in English for readability, summarizes a simple test for concluding that the level controller is in good condition. In the buffer tank process, as the level detector signal increases, suggesting a high water level in the tank, the level controller signal decreases to close the inlet valve and prevent the water level from climbing further. There is a test to make sure that the readings of the level detector gauge and level controller gauge have deviated in opposite directions, but there is no arbitrary (part three) action. Most of the rules in the knowledge base lack lisp code for arbitrary action; the only rules that have it are one rule that prints any conclusion about which fault is present and a rule that detects contradictions among conclusions added to the data base.

```
IF
(part 1)    gauge for level detector reads x
            and gauge for level controller reads y
(part 2) and x is opposite to y
THEN
(part 3)    do nothing special
(part 4) and add condition of level controller
            is good
```

Figure 6. A rule in the knowledge base.

Control Strategy. The control strategy for this module is simple. As statements about the observed data are added to the data base, they are checked against the rules. If a statement matches one pattern in part one of a rule, all the other patterns are checked to see if matching statements are present the data base. If all patterns can be matched, the test in part two of the rule is tried; if this succeeds, the code in part three is executed and the new statements in part four are added to a queue, where they await their turn to be added to the data base. All rules that can be applied will be applied eventually before module operation terminates.

Explanatory Facility. As was shown in Figure 5, the deductive fault analysis module explains its diagnosis by listing the premises from which the conclusion was derived. It is simply showing its deduction steps in reverse order, one rule at a time. This provides a good explanation provided the rules make sense to the plant operator.

- 1 fault may be inlet valve broken
- 2 tank may have failed
- 3 pipe 2 may have failed
- 4 pipe 1 may have failed
- 5 inlet valve may have failed
- 6 hi value of water source pressure may be principal effect of fault
- 7 hi value of flow through inlet valve may be principal effect of fault
- 8 hi value of flow from pipe 1 to pipe 2 may be principal effect of fault
- 9 hi value of flow from pipe 2 to tank may be principal effect of fault

Figure 7. Most likely faults causing the observed gauge readings.

inlet valve broken causes flow through inlet valve to be hi
 hi flow through inlet valve causes flow from pipe 1 to pipe 2 to be hi
 hi flow from pipe 1 to pipe 2 causes inlet flow gauge reading to be hi
 hi flow from pipe 1 to pipe 2 causes flow from pipe 2 to tank to be hi
 hi flow from pipe 2 to tank causes height of water in tank to be hi
 hi height of water in tank causes level detector signal to be hi
 hi level detector signal causes level detector gauge reading to be hi
 hi level detector signal causes high level alarm signal to be on
 hi level detector signal causes level controller signal to be lo
 lo level controller signal causes level controller gauge reading to be lo
 hi flow from pipe 2 to tank causes flow from tank to pipe 3 to be hi
 hi flow from tank to pipe 3 causes flow from pipe 3 to pipe 4 to be hi
 hi flow from pipe 3 to pipe 4 causes outlet flow gauge reading to be hi

Figure 8. Explanation of why fault may be inlet valve broken.

Causal Model Approach

Diagnosis. In this approach, the fault analysis module is given a model of the buffer tank process and a list of qualitative values for the observed data describing the state of the process, obtained from the monitor module. When the module is run, it reasons backwards from the observed data, with the help of the model of cause and effect relationships, to all the possible faults that might be hypothesized to explain some or all the data. After ordering them according to their likelihood, it displays the most likely faults. In the example, the module finds several possible explanations for the observed disturbances; these are shown in Figure 7.

The module can also explain why each of these faults is likely. When asked, it tells how a given fault can cause the observed gauge readings. For instance, when asked why the fault may be a broken inlet valve, it gives the answer in Figure 8.

Causal Model. The process is modelled as a system of interconnected components. Components are connected to each other by one or more fluents, the measurable quantities that would be

represented by variables in a mathematical model of the system. These quantities represent properties of matter and energy that pass from one component to another through a common point of physical contact. Fluents can be thought of as that which flows from the output of one component into the input of another. Each component (pipe, pump, reactor vessel, etc.) is well-understood and can fail in known ways. There are thus three kinds of objects in our model of a process: components, fluents and faults. These are connected together in a network.

There are three kinds of knowledge stored in the network. The first kind consists of the patterns of local disturbances caused by faults in individual components. The second consists of the manner in which disturbances propagate through components that are operating normally. The last consists of special knowledge about when certain chains of reasoning must be broken to avoid erroneous conclusions. This special knowledge will not be discussed further in this paper.

The information that is stored for each component consists of its English description, a list of the fluents that make up the inputs and outputs of the component, possible failure modes and disturbance propagation behavior. Take, for exam-

ple, the tank in Figure 2, shown in more detail in Figure 9. The tank T has three connecting fluents: the height H of its contents, the flow F2T from pipe 2, which is normally into T, and the flow FT3 into pipe 3, which is normally out of T. The tank can fail in two ways; it might develop a leak, which would lower the height of the water in the tank, and that would result in less flow into pipe 3 and more flow into the tank from pipe 2. Or something might suddenly fall into the tank, raising the height of the water to a level that turns on the high level alarm. The flow FT3 will increase and the flow F2T will decrease.

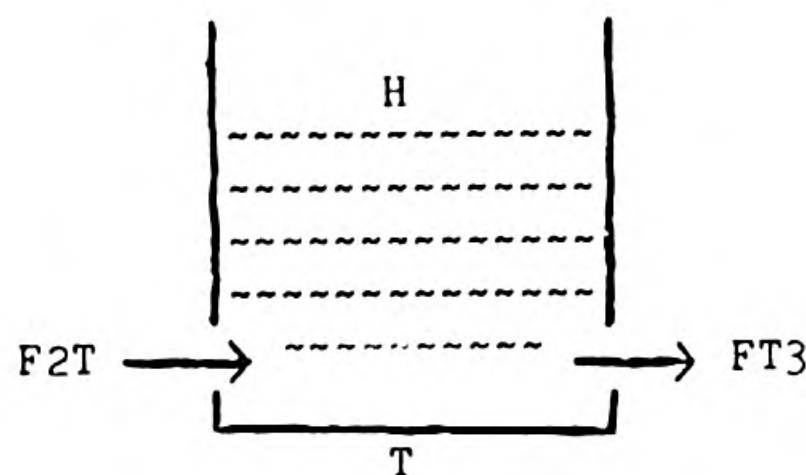


Figure 9. A tank with two pipe feeds.

If all faults are failures in individual process components, each fault can be represented by the pattern of qualitative changes it makes in a component's connecting fluents. Table 1 shows the disturbances resulting from a fault in the tank.

Fault	Qualitative change in fluent		
	F2T	H	FT3
leak in tank	hi	lo	lo
object in tank	lo	hi	hi

Table 1. Representation of faults in an individual component.

The propagation of disturbances through a normally operating component is modelled by rules stating how a disturbance in one connecting fluent can cause disturbances in the component's other connecting fluents. Since the only information required is whether the resulting disturbances are in the same direction or opposite from the one entering the component, these rules can also

be summarized in a table. Table 2 shows the propagation information for the tank. A plus (+) means that the affected fluent deviates in the same direction as the entering disturbance, and a minus (-) means that the deviation is in the opposite direction.

Disturbed fluent	Resulting disturbance		
	F2T	H	FT3
F2T		+	+
H	-		+
FT3	+	-	

Table 2. Disturbance propagation for tank T.

Note the lack of symmetry because the net flow through the tank is assumed to be from F2T to FT3. If the flow F2T into the tank increases, the height H of the water in the tank increases also. But if the height H increases, as when a large object accidentally falls into the tank, the flow F2T decreases because the back pressure from the water in the tank increases. This is a form of local negative feedback that can be inherent in a component.

Control Strategy. The input to this module is a list of observed fluents and their values. The module begins its analysis by tracing each observed fluent value backwards through the propagation rules, hypothesizing values for other fluents and storing hypotheses about each component, fault and fluent that can account for that observed fluent. Disturbances are classified as either hi or lo, depending on whether they are abnormally high or abnormally low compared to normal readings. Fluents with normal values are classified as ok. Alarms, because they respond to disturbances in one direction but not in the other, require special treatment; alarms that sound for high fluent values are hi when they are on, and ~hi when they are off. Alarms that sound for low fluent values are lo when they are on and ~lo when they are off.

Suppose that the inlet flow gauge reading is being traced and that the observed value is hi. Falcon traces component by component through the network to find all possible sources of this disturbance. It determines that the hi inlet flow gauge reading might be caused by a hi flow F12 from pipe 1 to pipe 2. This in turn may be caused by hi flow F2T from pipe 2 to the tank. According to Table 1, a hi F2T may be caused by a leak in the tank. According to Table 2, a hi F2T may be caused by a lo height H or a hi flow FT3 out of the tank. All these values become hypotheses that Falcon will evaluate in the next phase of the analysis.

Rating of Hypotheses. After the tracing phase, the module rates the fault hypotheses. It computes two numbers for each hypothesis; the first is the number of observed fluents explainable by the hypothesis, and the second is the number of observed fluents inconsistent with the hypothesis. These two numbers are used to sort the hypotheses in decreasing order of likelihood. One hypothesis is more likely than another if it explains more observed fluents. Among hypotheses that explain the same number of observed fluents, a hypothesis is more likely if it is inconsistent with fewer observed fluents.

After the hypotheses have been sorted, those that have the greatest likelihood are displayed. A fluent value hypothesis is displayed as a claim that a particular value for a particular fluent is enough to explain the observed data. A component hypothesis is displayed as a claim that a particular pattern of values for its connecting fluents is enough to explain the observed data. A fault hypothesis is displayed as a claim that a particular mode of failure in a particular component is enough to explain the observed data.

Explanatory Facility. When the most likely faults are displayed, they are numbered for easy reference. If the operator wants to know why the first fault is on the list, he can ask 'why 1', to which the module replies with a description of the consequences of the fault. If the fault is a known failure mode of some component, the pattern of fluent values for the outputs of that component are given first, and then the consequences of these values are traced to the observed fluents; an example of this was shown above. Hypothesized fluent values are explained in the same way. When a component is suspected to have failed, the explanation is the same except for minor changes in the description of the pattern of fluent values.

Conclusion

There are two versions of Falcon at present, depending on which of the two fault analysis modules is in use. With the knowledge-based fault analysis module, Falcon applies past experience, recorded as rules, directly to the data gathered from the alarms and gauges. The rules are similar to those used in the MYCIN program (Shortliffe [14]) for medical diagnosis, but they are applied in a forward direction, which is opposite to MYCIN. This is done because Falcon has most of the data it needs available directly from other computers already in the chemical plant through the monitor module. It will rarely ask operators for additional information. This is perhaps the simpler approach to building an expert system for alarm analysis in chemical plants, though collecting the rules from operators and engineers takes considerable effort.

The explanation given by this version of Falcon when asked why it has drawn certain conclusions, being a presentation of the deductive steps taken, is different in kind from the explanation given by the other version. It is not yet known which is more acceptable to plant operators.

In a sense, Falcon with the causal model-based fault analysis module combines the pattern matching/deductive approach to fault analysis and the causal model approach. Falcon traces disturbances through the causal model to find patterns of fluent values local to individual components. It matches these patterns against known patterns to draw specific conclusions about probable failure modes in those components. By rating the hypothesized patterns according to the amount of observed data that they can account for, Falcon is able to propose faults that are likely to be causes of the disturbances in the plant. Since a perfect match is not required, this approach to fault analysis can produce useful results from imperfect plant data. The method of choosing best matches on the basis of ratings and of data propagated through a network somewhat resembles the method by which INTERNIST (Pople [15]), a well-known expert medical system, diagnoses the presence of more than one disease in medical patients. Falcon also explains why it proposes certain faults by tracing their effects through the causal model.

The present Falcon program has many limitations which will eventually be overcome. It assumes that the monitored process is in a steady state. This may not be the case for all processes, and particularly not when the state is in transition, as in startup or shutdown. Some disturbances spread slowly through a process, disturbing observed values at different times. The causal model in Falcon should include time delay information so that intelligent diagnosis is possible while the disturbances are spreading; this may permit operators to act sooner to avert potential disasters. It may also be desirable to include probability information in both the causal model and the knowledge base so that highly implausible component failures will be dismissed in favor of multiple failures with higher probability.

Both versions of the Falcon program do well in tests on several processes and sets of observation data. With anticipated program changes to remove the limitations just noted, Falcon will provide a practical approach to on-line computer analysis of process data and diagnosis of component failures.

Acknowledgments

Many students are contributing to the Falcon project. They are, in alphabetical order, Bob Dondero, Joseph Elad, Apperson Johnson, John Lund, Senyo Opong, Ann Simon, Dan Stuart and Hongkoo Tae.

References

- [1] Berenblut, B. J., and Whitehouse, H.B. A method for monitoring process plant based on a decision table analysis. The Chemical Engineer (March 1977), 175-181.
- [2] Himmelblau, D. M. Fault Detection and Diagnosis in Chemical and Petrochemical Processes. Elsevier, Amsterdam-Oxford-New York, 1978.
- [3] Umeda, T., Kuriyama, T., O'Shima, E., and Matsuyama, H. A graphical approach to cause and effect analysis of chemical processing systems. Chemical Engineering Science, Vol. 35 (1980), 2379-2388.
- [4] Iri, M., Aoki, K., O'Shima, E., and Matsuyama, H. An algorithm for diagnosis of system failures in the chemical process. Computers and Chemical Engineering, Vol. 3 (1979), 489-493.
- [5] Andow, P. K. Real-time analysis of process plant alarms using a mini-computer. Computers and Chemical Engineering, Vol 4 (1980), 143-155.
- [6] Andow, P. K. and Lees, F. P. Process computer alarm analysis: outline of a method based on list processing. Trans. Instn Chem. Engrs, Vol. 53 (1975), 195-208.
- [7] O'Shima, E. Computer aided plant operation. Computers and Chemical Engineering, Vol. 7, No. 4 (1983), 311-329.
- [8] Lees, F. P. Process computer alarm and disturbance analysis: review of the state of the art. Computers and Chemical Engineering, Vol. 7, No. 6 (1983), 669-694.
- [9] Rasmussen, J. and Rouse, W. B., eds. Human Detection and Diagnosis of System Failures. Plenum Press, New York-London, 1981.
- [10] Nelson, W. R. REACTOR: an expert system for diagnosis and treatment of nuclear reactor accidents. Proceedings, AAAI-82, Pittsburgh, Pennsylvania (August 1982), 296-301.
- [11] Fox, M. S., Lowenfeld, S. and Kleinosky, P. Techniques for sensor-based diagnosis. Proceedings, IJCAI-83, Karlsruhe, West Germany (August 1983), 158-163.
- [12] Underwood, W. E. A CSA model-based nuclear power plant consultant. Proceedings, AAAI-82, Pittsburgh, Pennsylvania (August 1982), 302-305.
- [13] Yamada, N. and Motoda, H. A diagnosis method of dynamic system using the knowledge on system description. Proceedings, IJCAI-83, Karlsruhe, West Germany (August 1983), 225-229.
- [14] Shortliffe, E. H. Computer-Based Medical Consultations: MYCIN. Elsevier, New York-Oxford-Amsterdam, 1976.
- [15] Pople, H. E. The formation of composite hypotheses in diagnostic problem solving - an exercise in synthetic reasoning. Proceedings, IJCAI-77, Cambridge, Massachusetts (August 1977), 1030-1037.

James M. Galvin

University Of Delaware

ABSTRACT

In today's fast-paced, buy-now, sell-later, buy-again tomorrow, make-a-profit world of ulcer-laden executives, a computer programmer is often coerced into putting into production a software system that is not sufficiently tested. The correctness of the program or programs in question is rarely guaranteed. The problem can often be directly related to the amount of time allocated to the testing process. When the development of a system falls behind schedule it is often the testing of the program that suffers. A competent programmer tests a program on a well chosen set of test data. If no test case contradicts his confidence in the correctness of his program, testing stops. The problem is the test cases may reveal little or no information about whether the program is correct. If no attempt is made to evaluate the effectiveness or thoroughness of a set of test cases then the test cases can be a misleading sense of security. "A major reason for this omission is the lack of a generally agreed upon metric with which test cases can be measured." [1]

Mutation analysis is one such metric. It is a "radical new approach to determine test data adequacy which holds promise of being a major breakthrough in the field of software testing. ... Unlike previous work, program mutation assumes that competent programmers will produce programs which, if they are not correct, are almost correct. That is, if a program is not correct it is a mutant - it differs from a correct program by simple errors. Assuming this natural premise, a program P which is correct on test data T is subjected to a series of mutant operators to produce mutant programs which differ from P in very simple ways. The mutants are then executed on T. If all mutants give incorrect results then it is very likely that P is correct (ie., T is adequate). On the other hand, if some mutants are correct on T then either: (1) the mutants are equivalent to P, or (2) the test data T is inadequate. In the latter case, T must be augmented by examining the non-equivalent mutants which are correct on T: a procedure which forces close examination of P with respect to the mutants." [2] The Mutation Analysis

[1] Timothy A. Budd. Testing as an Inductive Process. University of Arizona Technical Report 80-27, 1980.

[2] Timothy A. Budd, Richard J. Lipton and Frederick G. Sayward. The design of a prototype mutation system for program testing. National

System is presented here with an example which demonstrates its usefulness and effectiveness.

INTRODUCTION

The final step in the development of any software system is the testing of the program or programs. Few programmers look forward to the testing process. Programmers, like ordinary people, do not like to admit to making mistakes. Testing implies there are mistakes in a program. After all, why test a program that is correct?

The purpose of testing is not usually to find things the programmer forgot or left out. More often one is looking for typing errors or misinterpretations of the specifications. The ultimate goal of testing is to demonstrate the correctness of a program, not the reliability or quality of a programmer. Although testing that consistently demonstrates the incorrectness of the programs of a particular programmer does not add to the lifetime of a programmer in a given environment. So why not just prove a program correct rather than testing it? Because proving a program correct is not always a decidable problem. In addition, it is extremely time consuming and tedious. Therefore, demonstrating the correctness of a program is usually considered sufficient.

But what is a correct program? Most people would respond that a program P is correct if it computes the desired function. This implies that you have explicit knowledge of the desired function F. But what form shall this knowledge take? If it is presented formally then one might as well interpret the specifications as the program itself and hence the question of correctness becomes simply one of equivalence. More likely, though, information concerning the correct function is almost never sufficiently detailed nor formalized. [3]

Therefore, in order to determine the correctness of P it is necessary to exercise P over the entire input space of F to be sure it generates

Computer Conference, Vol. 47, 1978.

[3] Timothy A. Budd and Dana Angluin, Two Notions of Correctness and Their Relation to Testing, Acta Informatica 18, 31-45, 1982.

the correct output space of F. Which is exactly the problem. In most cases, there is not enough time to do that kind of testing. Further, if one does not have explicit knowledge of the correct function F then the input space of F may not be determinable. On the other hand, the function F may be so complex that the output space is indeterminable. So the complete testing of the program P can be an unsolvable problem.

Suppose, however, we consider a set of programs N, which is sufficiently small, and assume that at least one program in the set N computes the desired function. The advantage of this approach is that a program can now be demonstrated to be correct by a process of elimination. By generating a test set for which all nonequivalent members of N fail on at least one point the program in question is shown to be correct.

Thus we have the following scenario. We are given a program P and a set of test cases which are believed to be sufficient to demonstrate the correctness of P. We assume P computes the correct answer for each of the given input items and that any program computing a different answer is incorrect. We generate the remainder of the set N, some closely related but distinct programs, and refer to the programs in N as the mutants of the program P. The mutants are formed by altering the program P in simple ways. We then run the mutants on the given test data and determine the number of live mutants and dead mutants. A live mutant is a program from the set N which computes the same results as the program P while a dead mutant is a program from the set N which computes different results than the program P.

If a large number of mutants compute the same results as the program P, there is no more reason to believe that the original program is correct than that any of the mutants are correct. Consider the analogous situation of the observation of a brown shoe. It gives us no more reason to believe that all ravens are black than that they are all red. On the other hand, if a majority of the mutants are dead then one can assert that the test data closely fits P and that P is likely to be a correct program.

Please note that the assertion is that P is likely to be a correct program, not that P is a correct program. The claim is that the test data is doing a good job of testing P, certainly better than that done by test data that fails to eliminate some mutants. The question is to what extent well tested implies being correct.

The effectiveness of this method of asserting the correctness of a program is based on a simple observation of the software development cycle; "namely, that most programmers are neither malicious nor incompetent, or at least those that are incompetent are quickly discovered by running their programs on almost any test data." [4] The

[4] Timothy A. Budd, Testing as an Inductive Process, University of Arizona Technical Report

assumption is that a good programmer will produce a program that if it is not correct is almost correct. More formally, the assumption is called the competent programmer hypothesis and stated specifically as follows:

A competent programmer, after giving the task sufficient thought and pursuing the normal process of programming and debugging, has most likely written a program that is either correct or almost correct, in that it differs from a correct program in simple ways. [5]

This is the basis for mutation analysis. We are given a program P and a set of test data believed to be sufficient to establish the correctness of P. We assume that P computes the correct answer on these inputs and that any program which computes a different answer is incorrect. We then ask whether these test cases increase our confidence in this program or could equally well be used to show the correctness of some closely related but distinct programs, the mutants of P.

COMPLEX ERRORS COUPLED

Certainly you would agree that if all possible errors a programmer might make in a program could be enumerated a mutant for each of the errors could be created. Then passing the mutation test would establish the correctness of the program in question. In other words, if the set N could contain a mutant for all possible errors and the test data was sufficient to distinguish a mutant from the given program by killing it, then the given program is necessarily correct.

Obviously it is unrealistic to assume that all possible errors can be enumerated. Therefore, mutation analysis uses a small set of carefully chosen transformations of the original program as a source of mutants. The transformations are in fact simple changes. Recall that the mutants are defined as being different from the original program in simple ways. How then does the detection of simple errors in a program ensure that complex errors do not also exist in the program?

Here again we lean on the competent programmer hypothesis. "The observation that programs are close to correct leads us to an assumption which makes the high frequency of simple errors very important:

The Coupling Effect: Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors.

80-27, 1980. Page 2.

[5] Timothy A. Budd, Testing as an Inductive Process, University of Arizona Technical Report 80-27, 1980. Page 3.

In other words, complex errors are coupled to simple errors. Rather than scamper after errors of undetermined character, the tester should attempt a systematic search for simple errors that will also uncover deeper errors via the coupling effect."[6]

MUTATION ANALYSIS

What follows is an introduction to the Mutation Analysis System itself. First an overview of the system, its implementation and a basic description of its operation. Second a discussion of mutants and how they are created is presented. Finally, each of the steps in the system is described.

Overview

The current implementation of the Mutation Analysis System is a Portable Suite. This new implementation is not yet complete. The Portable Suite is exactly what the name implies; the system can be easily ported to any other system. In addition, the new implementation is using LEX to create the mutants. This will allow extending Mutation Analysis to other languages; it will only be necessary to change the specifications to LEX.

The system operates on modules of a program. Ideally, all subprograms should be in their own files. The system is run on each of the files. As you would expect, the system provides a driver for the different modules it is presented. Currently, the system will recognize a subset of Fortran-77 (without I/O statements) with some restrictions.

Basically, the user has a program and a set of test data s/he wishes to evaluate. For best results, the program should be broken down such that the individual modules exist in their own files. These modules, and data to execute them on, is then translated into an internal form. This internal form is then used to create mutants which are then executed on the test data. The user must evaluate the living mutants and create test data that will distinguish the living mutants from the correct or original module. This new test data must be added to the system and the living mutants re-executed.

If it is determined that there is no test data that will distinguish the mutant from the original module and the mutant is an incorrect module, the user must rewrite the original module. In addition the testing process must be started over again. This new module must be translated and mutants must be recreated.

What is a Mutant?

In designing the system the construction of the mutants was viewed as a series of mutant operators to the original module. A mutant operator is a procedure that takes as input a module (in some form) and produces as output a set of mutant descriptions. A mutant description is a short encoding of the type of mutation and its location in the original module. It turns out that the mutants are actually interpreted rather than compiled and executed. This is because an interpreter based system allows greater control over program execution, which is important because many of the mutants are unstable; often resulting in infinite loops, zero divides or floating point interrupts.

Mutant operators can be divided into three groups. The first are source operand mutant operators. These alter the basic data objects constants, scalar variables and array references. Second there are several additional operators added to the usual Fortran operators. These operator mutant operators, along with the existing Fortran operators, are replaced by every other syntactically correct operator. Finally there are statement mutant operators which ensure that all blocks of code are executed, all statements are executed and all statement labels are unique.

There are three types of source mutant operators. There are nine operators which can be described by the form "replace each X with each distinct occurrence of Y" where X and Y range over the data objects. Another operator takes each constant and alters it slightly. For integers this means plus or minus 1, for reals plus or minus 10% or 0.01 if the value is zero, for logicals their complement and for characters the first character is replaced by the character's neighbors in the underlying ordinal scheme. Finally, all array names are replaced by all other array names of the same dimensionality.

The additional operators added to the operator mutant operators cannot appear in the original module, but are produced in the mutants. All operators are replaced by all other syntactically correct operators. In addition, unary operators are deleted or inserted wherever syntactically correct. Figure 1 describes the additional operators.

The statement mutant operators perform four basic functions. The initial statement of each basic block is replaced by a trap to insure execution of every statement. All statements are replaced by a CONTINUE to insure that each statement performs a useful function. All GOTO and arithmetic IF labels are changed to other labels in the program. The end of each DO loop is changed to some other label between the DO head and the end of the module. Optionally, the user can also have each DO loop replaced by a FOR loop so the loop may execute zero times.

[6]Richard A. DeMillo, et al. Hints on Test Data Selection: Help for the Practicing Programmer, Computer, April 1978. Page 35.

leftop, rightop (arithmetic or logical) - evaluate both operands but return only the left or right.

trueop, falseop (relational or logical) - evaluate both operands but return true or false.

twiddle (arithmetic) - denoted ++ or --, return plus or minus 1 for integer operands and plus or minus 0.01% or 0.01 whichever is greater for reals.

-ABS(X) (arithmetic) - always return the negative of X.

ZPUSH(X) (arithmetic) - return X if it is nonzero, otherwise a trap error occurs and the mutant is eliminated.

Figure 1

The Mutation System

The Mutation System is a five step process. Three of the five steps are repeated until the testing has been completed. In addition, there are two other optional steps, one of which has not been incorporated into the new system. Each of the steps is described below.

- (1) Before the testing can begin, the module to be tested must be parsed and converted to an internal form. This internal representation includes the code table, symbol table and other information. This step is performed once for each module to be tested.
- (2) Next the mutants must be created. In the new system, mutants are created for all of the available mutant operators. One of the optional steps is used to remove classes of mutants from the system before testing begins. This step is performed once for each module to be tested.
- (3) The testing process is now ready to begin. The user starts by entering the test data for the initial test. There is a program which inputs your test data and executes the module to be tested to determine the correct result. This result is then stored, along with the data, so that the execution of the mutants can be compared to it.
- (4) After the creation of the test data, each of the mutants is executed on each of the test data. If the results calculated by the mutant are equal to the results of the original module, the mutant is considered living and is saved. If the results differ, the mutant is considered dead and is discarded.

- (5) Following the execution of the mutants, the user can print a report of the living mutants. At this time, the user must add new test data that will cause the mutant to generate a different result than the original module, or determine that the mutant is equivalent to the original module.
- (6) Of the two optional steps, the one that works allows the user to remove classes of mutant operators from a test. This is best understood with an example, which is given later. Its most typical use is probably before testing begins. In the example presented here it was used after the testing process.
- (7) The last option is a step that allows the user to individually remove mutants deemed equivalent to the original module. When working, it will prompt the user whether to save individual mutants or kill them. This is not a recommended step to be performed first. Typically it would be used towards the end of testing to remove some of the clutter from the report mentioned above.

```
subroutine triang(i,j,k,match)
  readonly i,j,k
  output match
  if (i.le.0.or.j.le.0.or.k.le.0) goto 500
  match=0
  if (i.ne.j) goto 10
  match=match+1
10  if(i.ne.k) goto 20
  match=match+2
20  if(j.ne.k) goto 30
  match=match+3
30  if(match.ne.0) goto 100
  if(i+j.le.k) goto 500
  if(j+k.le.i) goto 500
  if(i+k.le.j)goto 500
  match=1
  return
100 if(match.ne.1) goto 200
  if(i+j.le.k)goto 500
110 match=2
  return
200 if(match.ne.2) goto 300
  if(i+k.le.j) goto 500
  goto 110
300 if(match.ne.3) goto 400
  if(j+k.le.i) goto 500
  goto 110
400 match=3
  return
500 match=4
  return
end
```

Figure 2

THE EXAMPLE

The example presented is based on the subprogram in Figure 2, which was taken from a paper by Lori Clarke and John L. Woods entitled "Program Testing Using Symbolic Execution." This subprogram determines whether or not the three values passed (i, j, k) represent the sides of a triangle. Match is output as follows: 1 if three values represent a scalene triangle, 2 if they represent an isosceles triangle, 3 if they represent an equilateral triangle or 4 if they are not a triangle.

The first thing you should notice is the extensions to the Fortran declarations. The system lets you tell it if a variable is not supposed to change throughout the lifetime of the module. In addition, you can indicate variables which are strictly for output. The information is helpful for killing mutants. It aids in the detection of undefined variables and the modification of variables which are intended to be call-by-value.

In all of the examples shown here, the "%" is the prompt used by my computer. Items in *italics* are typed by the user. Everything else is a result of the Mutation System, except the computer prompt.

The first step in the analysis is to parse the original module. (See figure 3.) *Parse* is the command which takes the module as input and translates it into an internal form. The *-n* is used to give a name to the experiment. In this case I chose *ex*. Next the name of the file with the module to be tested is given, *prog.f*. Finally, the *-di* is one of three possible debugging switches that can be used. The *-di* is used

to print out information about variables. The *-dc* switch will print out the code as it is being generated. The *-d* option will do both.

The warning messages which printed are just that. They should be self-explanatory. The remainder of the information which printed is a result of the *-di* switch. Each variable is printed along with its known attributes. I have not found any real value for the information. The *-dc* switch prints out a very cryptic version of the code. The code is not documented and as such the output is of no value.

The next step is to create the mutants. (See figure 4.) From this step forward it is only necessary to give the experiment name. The remainder of the system only reads the internal form of the module. The output generated summarizes the types of mutants which were created. The information is useful for determining the most common type of statement or expression in your program. As of the writing of this paper, mutant levels were not documented. Since there were no old mutants, the total number of new mutants created is exactly the total number of mutants.

This completes the initialization necessary to run the remainder of the system. The user must now input a set of test cases and collect live mutants. Then the user repeatedly adds test cases and executes the remaining mutants on the test cases until all mutants die or the remaining mutants are equivalent to the original module.

The evaluation begins with the inputting of the initial set of test cases. For the purposes of demonstration, I chose a set of test data that does not exercise the program very well. This will leave a large number of living mutants to create more test data for. In particular, I

```
% parse -n ex prog.f -di
Warning, variable i      given type integer
Warning, variable j      given type integer
Warning, variable k      given type integer
Warning, variable match  given type integer
i      is the following:
read only
input
parameter
j      is the following:
read only
input
parameter
k      is the following:
read only
input
parameter
match is the following:
output
parameter
```

Figure 3

```
% makmut ex
Mutant maker statistics:
  484 new mutants were made.
  484 total mutants now exist.
   49 level 1 mutants made.
  219 level 2 mutants made.
   45 level 3 mutants made.
  171 level 4 mutants made.
New types made:
   13 continue replacements
   32 logical replacements
  185 absolute value and zpush insertions
   54 left, right, true and false replacements
   29 short integer constant replacements
   45 operator replacements
  126 scalar replacements
Old types:
None.
```

Figure 4

```

% rdtest ex
Enter initial values for test case number 1.
Enter the value for i
1
Enter the value for j
2
Enter the value for k
3
match = 4
Are these the correct results?
y
Any more test cases?
y
Enter initial values for test case number 2.
Enter the value for i
2
Enter the value for j
2
Enter the value for k
3
match = 2
Are these the correct results?
y
Any more test cases?
y
. . .
Any more test cases?
n

```

Figure 5

selected four inputs for the first set of test data; one for each of the possible outputs of match. Figure 5 is a sample of what inputting test data is like.

After inputting the test cases, each of the mutants must be executed on each of the test cases. Figure 6 is a sample of the output that can be generated when the mutants are executed.

This output is controlled by the `-t` switch on the command line. If absent, no notification is given of the status of individual mutants. This is a recommended option for large numbers of mutants. It should be obvious that this output is voluminous. If you would like occasional notification, just so you know the program is doing something, the `-t` option can be followed by an integer which indicates how often to print the status of a mutant; `-t N` means print notification every N'th mutant. The `-s` switch is interesting. It prints out statistics on the number and kind of mutants that died. In the listing above you will find one example of each of the statistics summarized at the end.

The four inputs I selected hardly exercised the program at all. A mortality rate of almost 40% is not meant to be typical. A program which has been adequately tested initially, should kill off about 80% of the mutants the first time. This figure is known to be typical to the author of the

system.

Since almost two-thirds of the mutants survived, there is quite a bit of work remaining to build confidence in the module being tested. What we need now is a list of the mutants that survived so we can construct test data that will distinguish our original module from the mutants. The final step in the mutation process is to print a report of the live mutants and the test data used to execute them. Figure 7 demonstrates this command.

The output is clearly voluminous (truncated for brevity). Note carefully the syntax. All living mutants are clearly indicated in the output. The actual executable lines of the module are indicated by a line number preceding a `"|"`. Lines which begin with a `">"` are living mutants. Mutants differ from the original module in exactly one statement. To create a mutant simply copy the original module and substitute one of the lines beginning with a `">"` for the appropriate line.

Following the listing of the mutants was a list of the test cases the mutants were executed on (truncated for brevity). This list is included when the `-c` switch is set on the command line. The `-d` switch is supposed to print out debugging information before each mutant, but it does not seem to work.

The appropriate thing to do at this time is construct more test data. In this example, it is possible to kill off all the mutants. Several have to be killed manually, however. In particular, all the mutants consisting of the operators ZPUSH, ABS and -ABS are equivalent mutants. This is because the first executable statement of the module ensures that only positive values are allowed to continue in the rest of the module. Therefore, there is no way to construct test data to eliminate these mutants. This is an example of the use of the optional command to delete mutants created by classes of mutant operators.

In addition, statement 6 has a mutant of the form `match = 1`, statement 9 has one of the form `match = 2` and statement 12 has one of the form `match = 3`. These cannot be killed because the assignments are precisely the value of match at that time. They are also equivalent mutants. Finally statement 31 has two mutants which are also equivalent. The first is changing the conditional to `i + i .le. j` and the second is changing it to `k + k .le. j`. These are equivalent because the only way to be at that line of code is if `i` is equal to `k`. Since the original conditional is testing `i + k`, there is no test data to eliminate these mutants. Sufficient test data will kill all other mutants.

After the construction of more test data, the process need only be repeated from the inputting of the test data. The inputting of test data is cumulative. This means that previous data is not lost when additional data is entered.

```

% mutate ex -t 1 -s
experiment name is ex
  Mutant number          1
      if (.true..or.k.le.0)
killed on test case      2 cause:
Error number -1 - results differ
. . .
  Mutant number          3
      if (.false..or.k.le.0)
survived all test cases
. . .
  Mutant number          40
      if (match.le.0.or.j.le.0.or.k.le.0)
killed on test case      1 cause:
Error number 9 - Undefined variable used
. . .
  Mutant number          50
      i = 0
killed on test case      1 cause:
Error number 30 - Attempt to modify read only variable
. . .
  Mutant number          369
      continue
killed on test case      2 cause:
Error number 7 - Time limit, too many instructions executed
. . .
Mutant statistics:
  186 died.      298 lived.  For a mortality rate of 38.4%

Individual statistics on mutants:
Results differed on          155
Time limit                    1
Undefined variable used       6
Attempt to modify read only variable 24

```

Figure 6

EVALUATION

The following evaluation is from an implementation and theoretical point of view. The conclusions contain an evaluation from a users point of view. In this evaluation answers to the following three questions will be examined. Note that answers to the first two questions should trade off in much the same way that time and space do in algorithms. Also, the examination of the third question is the ultimate evaluation of any program testing system.

- (1) What is the cost of running/using the mutation system?
- (2) What is gained from running/using the mutation system?
- (3) Are there incorrect programs that pass the system's test?

Cost of the Mutation System

The most obvious cost of running the mutation system is the number of mutants generated which

must be executed on the set of test data. Table 1[7] gives, for 16 typical programs, the number of mutants generated along with some other metrics for the programs:

N is the number of executable statements in the program.

M is the number of mutants generated.

X is the number of references made to constants, scalar variables and arrays.

Y is the number of references indicated above which are distinct.

E is Halstead's Effort metric.

V is McCabe's complexity metric.

[7] Timothy A. Budd, Testing as an Inductive Process, University of Arizona Technical Report 80-27, 1980. Page 14.

% report ex -c -d

ex

```

subroutine triang(i,j,k,match)
integer i,j,k,match
readonly i,j,k
output match
input i,j,k
1| if (i.le.0.or.j.le.0.or.k.le.0)
> | if (.false..or.k.le.0)
> | if (.false.)
> | if (.false..or.j.le.0.or.k.le.0)
> | if (i.le.0.or..false..or.k.le.0)
> | if (i.le.0.or.j.le.0.or..false.)
> | if (inc(i).le.0.or.j.le.0.or.k.le.0)
> | if (dec(i).le.0.or.j.le.0.or.k.le.0)
> | if (abs(i).le.0.or.j.le.0.or.k.le.0)
> | if (zpush(i).le.0.or.j.le.0.or.k.le.0)
> | if (i.le.-1.or.j.le.0.or.k.le.0)
> | if (i.le.1.or.j.le.0.or.k.le.0)
> | if (i.lt.0.or.j.le.0.or.k.le.0)
> | if (i.eq.0.or.j.le.0.or.k.le.0)
> | if (i.le.0.or.inc(j).le.0.or.k.le.0)
> | if (i.le.0.or.dec(j).le.0.or.k.le.0)
> | if (i.le.0.or.abs(j).le.0.or.k.le.0)
> | if (i.le.0.or.zpush(j).le.0.or.k.le.0)
> | if (i.le.0.or.j.le.-1.or.k.le.0)
> | if (i.le.0.or.j.le.1.or.k.le.0)
> | if (i.le.0.or.j.lt.0.or.k.le.0)
> | if (i.le.0.or.j.eq.0.or.k.le.0)
> | if (i.le.0.or.j.le.0.or.inc(k).le.0)
> | if (i.le.0.or.j.le.0.or.dec(k).le.0)
> | if (i.le.0.or.j.le.0.or.abs(k).le.0)
> | if (i.le.0.or.j.le.0.or.zpush(k).le.0)
> | if (i.le.0.or.j.le.0.or.k.le.-1)
> | if (i.le.0.or.j.le.0.or.k.le.1)
> | if (i.le.0.or.j.le.0.or.k.lt.0)
> | if (i.le.0.or.j.le.0.or.k.eq.0)
> | if (j.le.0.or.j.le.0.or.k.le.0)
5 more mutants.
2| $goto 500
3| match = 0
4| if (i.ne.j)
> | if (zpush(i).ne.j)
> | if (i.ne.abs(j))
> | if (i.ne.zpush(j))
> | if (i.lt.j)
5| $goto 10
6| match = match+1
> | match = abs(match)+1
> | match = -abs(match)+1
> | match = match+2
> | match = abs(match+1)
> | match = zpush(match+1)
> | match = 1
> | match = i+1
> | match = j+1
7| 10 if (i.ne.k)
> | 10 if (inc(i).ne.k)
> | 10 if (dec(i).ne.k)
> | 10 if (abs(i).ne.k)

```

```

> | 10 if (-abs(i).ne.k)
> | 10 if (zpush(i).ne.k)
> | 10 if (i.ne.inc(k))
> | 10 if (i.ne.dec(k))
> | 10 if (i.ne.abs(k))
> | 10 if (i.ne.-abs(k))
> | 10 if (i.ne.zpush(k))
> | 10 if (i.lt.k)
> | 10 if (j.ne.k)
> | 10 if (match.ne.k)
> | 10 if (i.ne.j)
> | 10 if (i.ne.match)
8| $goto 20
9| match = match+2
> | match = inc(match)+2
> | match = dec(match)+2
> | match = abs(match)+2
> | match = -abs(match)+2
> | match = zpush(match)+2
> | match = match+1
> | match = match+3
> | match = abs(match+2)
> | match = zpush(match+2)
> | match = match
> | match = 2
> | match = i+2
> | match = j+2
> | match = k+2
> | match = match%2
> | match = match*2
> | match = match**2
10| 20 if (j.ne.k)
> | 20 if (zpush(j).ne.k)
> | 20 if (j.ne.abs(k))
> | 20 if (j.ne.zpush(k))
> | 20 if (j.lt.k)
> | 20 if (i.ne.k)
11| $goto 30
. . .
43| end
Test case 1
variable i
1
variable j
2
variable k
3
variable match
4
Test case 2
. . .

```

Figure 7

N	M	X	Y	E	V
12	2580	103	21	32033	1
13	317	27	8	4071	5
17	386	32	8	6928	4
17	634	45	9	15246	7
24	2716	72	40	17565	7
26	646	40	11	16270	9
33	859	55	13	41819	12
33	23382	407	53	249701	1
56	3657	129	23	138939	9
66	2425	115	15	170492	10
67	5230	158	28	189585	15
71	2888	135	16	166715	11
98	8457	227	32	365825	22
112	16380	237	68	320331	26
277	34657	545	63	3024488	122
514	120000	1138	93	19267409	113

Table 1: Number of mutants generated versus program size

The actual number of mutants generated correlates very highly with the product of the number of references made to constants, scalar variables and arrays and the number of these references which are distinct.

This high correlation is probably due to the "replace every member of X with a member of Y" nature of the data mutations and the fact that data mutations account for an average of 82% of all mutants generated (statement mutants account for about 13% and operator mutants for the remaining 5%). [8]

It is important to note however, that the large number of mutants generated is not as much of a problem as one might think at first. It turns out that about 80% of all mutants die the first time they are encountered, no matter how good or bad the test data is; and they more often than not do not execute in their entirety. The reason is because there are eight ways a mutant can die:

- (1) by computing the wrong answer, which in the first few test cases is the least likely way a mutant will die.
- (2) by receiving an arithmetic fault.
- (3) by computing a subscript index out of bounds.
- (4) by executing a trap statement (used in path analysis, did we reach this section of the code).

[8] Timothy A. Budd, Testing as an Inductive Process, University of Arizona Technical Report 80-27, 1980. Page 14.

- (5) by referring to an undefined variable.
- (6) by attempting to divide by zero.
- (7) by running for too long.
- (8) by attempting to change a read only variable.

It is not until after the first few test cases that mutants begin to consistently execute in their entirety and die because they compute an answer different than that computed by the original program.

Further, it has been observed that in achieving the goal of eliminating non-equivalent mutants, about twice as many mutant executions are performed as there are mutants generated. If we eliminate the equivalent mutants, those that survive all test cases, the average lifespan of a mutant is only 1.5 test cases. Typically, the last 50% of the test cases are used to eliminate the last 2-10% of the mutants.

Another cost of mutation analysis is the time it takes to run. It is not necessarily the total amount of time spent on mutation analysis but the time between iterations. There can be a significant delay between when the first test cases are being run on the initially generated mutants and when the second test case run can be started. Obviously, when there are several thousand mutants to execute there is not a great deal of interaction between the programmer and the mutation system. The suggestion has been made, however, that at least the first iteration is best accomplished overnight, in a batch processing environment.

Gained From the Mutation System

The most important gain from using mutation analysis is the confidence in the correctness of a given program it establishes. Given a program and a set of test cases it objectively and consistently measures how well the test cases exercise the program. With this information the correctness of a program is very well established.

It should be pointed out, though, that like most testing methods, mutation analysis is not infallible. Consider, for example, the two programs in figure 8.

The programs both compute the same answer on the data set {0 0 0 0, 1 -3 -2 8, -2 5 3 -6} which satisfies the following testing criteria:

- (1) Every statement, branch and path is executed.
- (2) Every expression is, at times, both positive and negative, as well as identically zero.
- (3) No operator can be replaced by any other operator.
- (4) All mutants in the mutation system are eliminated.

"This example shows that even in the very restrictive class of straight line FORTRAN programs with only addition and subtraction tremendous difficulties can be encountered. It is possible that these difficulties cannot be eliminated by any non-exhaustive testing methods. What then saves us from a morass of intractability?

"The saving grace is that programmers are usually not adversaries, hiding inscrutable mistakes in obscure locations. Most errors a programmer makes are simple in form, well understood and classifiable. Furthermore, we are willing to live with something less than assurances of total correctness. If we observe that the program works correctly for a large number of well designed inputs and that the program has been reasonably well exercised, then we are usually willing to believe it is correct (always keeping in mind the small possibility we may be wrong). The fact that

```
FUNCTION E(A,B,C,D)
  F = B + C
  G = A - F
  E = G + D
END
```

```
FUNCTION E(A,B,C,D)
  F = B + A
  G = A - F
  E = G + D - B
END
```

Figure 8

these goals are not perfect but are attainable is why testing has been and will continue to be established practice among programmers." [9]

Another gain of using mutation analysis is somewhat more subtle. In theory, a major stumbling block to the application of mutation analysis is the problem posed by mutants that are equivalent to the original program. In practice, equivalent mutants are a nuisance, but for an entirely different reason. It is not that equivalent mutants are difficult to discover, but that they are so prevalent and simple minded that they get in the way of the more important aspects of testing. In spite of all that, though, mutants are a valuable resource providing the greatest insight into the functioning of the program.

Undetectable Errors

One example of a program for which mutation analysis would fail has been given above. Programs such as that however, are rare, especially in light of the competent programmer hypothesis. Notice that the mutation method exists quite independently of the mutation system. The selection of the mutant operators in the system is not meant to remain fixed. Further experience with the system will undoubtedly suggest new operators to be incorporated into the system.

Also, it is very conceivable that an alternate system could be constructed using the mutation method but with a different set of mutant operators. Such a system could behave radically different from the previous system and in fact detect quite a different class of errors.

Clearly, the performance of the mutation system is limited only by the mutant operators with which it operates. Given its effectiveness one could envision a mutation system where a programmer could select classes of errors to look for and create in a given program rather than being burdened with the selection of which mutant operators to apply.

CONCLUSIONS

In this paper I have presented a metric which can be used to evaluate the thoroughness with which a given set of test cases exercises a program. The metric is called mutation analysis and consists of comparing the given program against a number of alternative, or mutant, programs.

The Mutational Analysis System is very effective. Unfortunately, due to the conversion to a Portable Mutation Suite, it is unstable and incomplete at the moment. However, when complete, it will be easy to use. I think that some of the output could be changed for the better. It would be nice if the inputting of the test cases did not

[9] Timothy A. Budd, Testing as an Inductive Process, University of Arizona Technical Report 80-27, 1980. Page 15.

have to be so voluminous. The prompts get rather distracting after a while. Also, the report that prints the mutants and the test data should have the data printed in some kind of table. It is very difficult to read the listing. Finally, the -dc switch for parse should print the code in a form suitable for human consumption. Either that or it should be documented as intended for use by the system designer.

The most obvious foible is not dealing with input and output. In addition, there are several Fortran constructs that are not dealt with adequately. Consider the following.

- (1) Fortran DO loops are required to have integer bounds.
- (2) DATA statements do not work.
- (3) DOUBLE PRECISION and COMPLEX data types are not allowed.
- (4) EQUIVALENCE statements are particularly unstable.
- (5) There is also the problem of communication between modules. COMMON blocks are out for this reason and the system does not allow the testing of more than one module at a time.

I should point out that what does work, works very well. The mutation method has been shown to work as well as if not better than seven other testing methods.[10] It is a method that should be looked into more carefully, both empirically and theoretically.

Mutation analysis has the one real disadvantage of not being able to be theoretical proven. This is because it is based on two assumptions, the "competent programmer hypothesis" and the "coupling effect", neither of which can be proven. However, mutation analysis has the one real advantage in the fact that it works. Although they have not been presented here, many empirical studies have been done which show that it performs at least as well as any other testing method. In most case, in fact, it performs significantly better because, as shown above, it can achieve the goals of more than half dozen different testing methods combined. I am sure that it would be more widely accepted and used if it could be proven to effectively demonstrate the correctness of a given program.

REFERENCES

- [1] Budd, Timothy A., Richard J. Lipton, Frederick G. Sayward and Richard A. DeMillo. "The Design of a Prototype Mutation System for Program Testing". Conference Proceedings, National Computer Conference, Vol. 47, 1978.
- [2] Budd, Timothy A. Testing as an Inductive Process. University of Arizona Technical Report 80-27, 1980.
- [3] Budd, Timothy A., Richard A. DeMillo, Richard J. Lipton and Frederick G. Sayward. "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs". ACM SIGPLAN, 1980.
- [4] Budd, Timothy A. and Dana Angluin. "Two Notions of Correctness and Their Relation to Testing". ACTA Informatica. 1982.
- [5] DeMillo, Richard A., Richard J. Lipton and Frederick G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer". COMPUTER, April 1978.

[10] Timothy A. Budd. Testing as an Inductive Process. University of Arizona Technical Report 80-27, 1980.

COMPUTER REQUIREMENTS FOR NONTRADITIONAL CONTROL

Ronald K. Pearson

E. I. du Pont de Nemours & Co., Inc.
Experimental Station
Wilmington, Delaware 19898

Abstract

Most process control computer systems now available have been developed to implement traditional control system designs. While they handle this task quite well, these systems are generally not flexible enough to implement nontraditional control system components like the Kalman filter. These systems frequently incorporate microprocessors with enough computing power to implement nontraditional control algorithms, suggesting that the limitations are inherent in the software. To illustrate these limitations and suggest some possible solutions, this paper examines the overall software structure necessary to implement a multivariable control algorithm recently proposed by T. S. Chang and D. E. Seborg to handle inequality constraints on process variables.

Introduction

Traditionally, the control of continuous industrial processes has been based on the three-term (proportional, integral and derivative) controller, either used alone or in various combinations to construct more complex control systems. Originally, this approach was chosen because hardware limitations made more complex building blocks impractically expensive. However, the advent of the microprocessor has changed this situation markedly, providing inexpensive compact hardware that can implement arbitrarily complex control algorithms. While powerful microprocessors like the Motorola 68000 and the Intel 8086 have been employed in commercial controllers, they have generally been used to make traditional control systems cheaper and easier to configure rather than to provide new features. A major selling point of many of these systems is "user friendliness," usually stated as "you don't have to be a programmer to use this system." While this is a significant advantage in developing conventional control schemes, it usually also means that the software is configured so that the computing power of the processor is unavailable for anything other than the specific functions built in by the manufacturer.

This paper is a reasonably detailed examination of the software structural requirements for

a hypothetical system of sufficient flexibility to implement nontraditional model-based control systems, typified by an approach recently proposed by T. S. Chang and D. E. Seborg¹. A point of reference is first established by briefly summarizing the nature of traditional control and examining the overall structure of a typical commercial microprocessor-based controller. Next, the Chang and Seborg algorithm is described to illustrate the computational requirements for a typical model-based control algorithm. Finally, a flexible software structure is proposed that would facilitate the implementation of both traditional control systems by nonprogrammers and nontraditional control systems by more experienced users.

Traditional Controller Structures

Traditional control system design is based on a relatively small set of standard building blocks or "control idioms" in the terminology of Bristol². The standard idioms include a variety of single-input, single-output dynamic transfer function blocks like PID controllers and lead-lag compensators, and static coordination blocks with two or three inputs and outputs like static decouplers and high/low limit selectors, used to coordinate the dynamic elements and determine the overall structure of the control system. Most of these building blocks have one or more free parameters that can be adjusted over a finite range of values to adapt the behavior of the composite control system to the process being controlled. For example, in the lead-lag compensator, the numerator and denominator time constants that define the lead-lag transfer function can each be adjusted over a range of positive values. The resulting transfer function introduces no phase shift at either dc or high frequencies but introduces either a phase lead or a phase lag at intermediate frequencies, depending on which time constant is larger. The effect this will have on control system performance will depend on the overall structure of both the control system and the process being controlled.

Most commercial control hardware was designed to facilitate the development of control systems by providing reliable, standardized implementations of the most popular control idioms. Because they evolved when all signal processing was done by either analog electronic or pneumatic

hardware, these control idioms are computationally simple. Further, because typical process time constants are on the order of minutes, they represent a minimal computational load. Consequently, commercial control system vendors have chosen to use the computing power now available from microprocessors to consolidate earlier products that each provided one of these control idioms into a more flexible package that could be configured to function as any one of several different idioms or simple combinations thereof. In addition, the microprocessors currently employed have sufficient excess computational capacity to allow the inclusion of crude interactive configuration programs as part of the controller itself.

A simple data flow diagram³ is shown in Figure 1 illustrating the global structure of a typical microprocessor-based configurable controller. In the newest models, the hardware includes a 16-bit microprocessor that implements all of these processing steps, enclosed in a package that occupies less than a cubic foot of space in a standard relay rack. As indicated in the data flow diagram, the control system communicates with the outside world through five standard interfaces. At the lowest level, process variables like pressure, temperature and flow rate are input from electronic or pneumatic sensors. These are processed by the control algorithm that implements the selected idiom(s) and control signals are output to an actuator (typically a control valve). Moving up in authority, the operator communicates with the controller through a front panel, visually monitoring the variable being controlled, and adjusting the controller set point as needed in normal operation. In response to anomalous situations, the operator may choose to put the controller in manual, setting the output signal to the control actuator directly. At the next level in the user hierarchy, an instrument mechanic is periodically called in to retune the controller, adjusting free parameters in the control algorithm through an internal configuration and tuning keypad to achieve (hopefully) better performance. At the highest level of authority in the user's facility, the control engineer who installs or reconfigures the controller can execute the configuration routine from this keypad to specify how the available control idioms are to be connected together to form the overall process control system.

In addition to these five user interfaces, a sixth data flow is shown in Figure 1, connecting the manufacturer of the controller to the options menu that defines the control idioms and tuning parameter limits. The essential point here is that the variety of control idioms available to the control engineer and the range over which their free parameters may be adjusted by the instrument mechanic are decided in advance by the manufacturer and are totally outside the end user's control.

Nontraditional Control Approaches

While the control system development approach outlined above is the most common in the chemical process industry, other approaches have been proposed and successfully applied, especially in academia and other industries. One particularly promising class of methods imbeds a dynamic model of the process in the controller, either to estimate internal variables that are not measurable directly⁴, to account for the effects of unmeasurable disturbances on the process⁵, or to minimize the deviations of process variables from their set points subject to constraints on both manipulated variables and internal process variables¹. Because of their model dependence, these methods are not advocated for the design of centralized controllers for an entire process, but they can be quite useful in improving control of critical process components. Unfortunately, it is usually either impossible or prohibitively difficult to implement these control strategies with configurable controllers that restrict designs to simple combinations of standard idioms like PID controllers, square root extractors and high/low limit selectors. It is not difficult to see, however, that these techniques could easily be implemented in the same hardware if software provisions were made for customized control algorithms.

To illustrate this point, consider the constrained control algorithm of Chang and Seborg¹. Its basis is the discrete-time dynamic process model

$$x(k+1) = Ax(k) + Bu(k) + Dw(k) \quad (1)$$

$$y(k) = Cx(k) + v(k) \quad (2)$$

where x is an n -dimensional vector of variables describing the state of the process at any fixed time, u is an r -dimensional vector of manipulated process variables, d is a p -dimensional vector of unmeasurable process disturbances, y is an m -dimensional vector of measured process variables used for control and v is an m -dimensional vector of (random or systematic) measurement errors. The argument k associated with all of these vectors indicates they are sampled at uniformly spaced discrete time instants $k=1,2,\dots$. Practically, such models may be derived either empirically via time series analysis techniques⁶ or by developing a nonlinear first principles (e.g., heat and mass transfer) model, linearizing it about the assumed operating conditions and discretizing the result⁷. Although they are gross oversimplifications of the process dynamics, these models can be reasonably successful in describing modest deviations of process variables about a given set of steady-state operating conditions at the sampling instants.

The control approach considered by Chang and Seborg uses the known or computable process variables in equations (1) and (2) to predict the future values of the process state and output variables under an assumed control strategy.

That is, they assume $d(k)=0$ and $v(k)=0$ for all k and use equations (1) and (2) to predict the values for $x(k+1)$ and $y(k+1)$ that a particular choice of manipulated variables $u(k)$ would generate if applied at the k^{th} sampling time. From this, it is possible to select a control vector $u(k)$ that will minimize the deviations of process variables from their set points. Specifically, Chang and Seborg choose $u(k)$ to minimize the performance index $J(k)$, defined as

$$J(k) = \sum_{i=1}^n w_i p_i(k+1) + M_i \sum_{j=1}^k e_i(j) \quad (3)$$

where $p_i(k+1)$ is the predicted deviation in the i^{th} state variable $x_i(k+1)$ from its set point resulting from the control vector $u(k)$ and $e_i(j)$ is the actual deviation in $x_i(k)$ incurred at time j from the control policy in effect until then. The constants w_i and M_i are weighting factors and may be viewed as tuning parameters for this control algorithm.

Practically speaking, the most important feature of Chang and Seborg's algorithm is that the performance index $J(k)$ is minimized subject to physically motivated constraints on the internal state variables $x(k)$, the measured process variables $y(k)$ and the manipulated variables $u(k)$ (e.g., no control valves are more than 100% open nor less than 0% open). Due to the particular choice of performance index, this optimization can be reduced to a linear programming (LP) problem with $2n+r$ variables and $3n+2m+r$ inequality constraints¹. Consequently, it can be solved efficiently by standard techniques like the simplex method⁸, yielding a control algorithm that can be executed in real time at realistic sampling rates by current generation microprocessors. It is clear, however, that this algorithm is not well suited to implementation with standard configurable controllers. (This might be possible for sufficiently small m , n and r through some clever combination of gain blocks and high/low limit selectors, but it is analogous to writing Fast Fourier Transforms in COBOL - the task is clearly difficult enough to discourage use of the method.)

A Hypothetical Controller Structure

As an alternative, Figure 2 is a data flow diagram for a hypothetical controller flexible enough to easily implement nontraditional control algorithms like Chang and Seborg's. This controller communicates with the process through standard input and output data conversion routines connected to the same sensors and control actuators as the configurable controller does. The communication interfaces with the operator and instrument mechanic are essentially the same as those of the standard controller, but the software is designed to facilitate the incorporation of custom control algorithms. The only visible differences to the operator are that the front panel can display unmeasurable state variables computed internally by the control algorithm

(e.g., average molecular weight in a polymerization reactor) and that front panel controls can be used to enter set points for such variables, depending on how the algorithm is structured. Similarly, there may be more tuning parameters for the instrument mechanic to adjust than there are in a standard configurable controller. It is important to note, however, that the choice of internal variables that can be displayed on the front panel, set points that can be adjusted from the front panel, and tuning parameters that can be adjusted from the tuning keypad is left entirely to the control engineer who designs the algorithm. Unlike the standard configurable controller, these choices, along with the actual control algorithm to be executed, are specified off-line by the end user on a host development system.

Internally, the software is structured so that all interfaces with the control algorithm are data stores, implemented as reserved areas in random access memory for measured variable values, set point values, tuning parameters, internally computed process variables (in the scratch pad) and outputs to the control actuators. This makes the problem of developing custom control algorithms analogous to that of developing custom I/O drivers for general purpose computers. Specifically, control algorithms would be written in a higher level language on a host development machine, subject only to the constraints of controller sampling rates, memory size and data store restrictions (e.g., control algorithms can read values from the measured variables table but not write to it; specific memory locations in the table correspond to physical terminal locations on the controller package; etc.). Indeed, one of the most significant features of this design is that the entire configuration task has been removed from the controller, freeing more of the microprocessor's computational capacity for control activities.

One of the major advantages of this controller structure over that of standard configurable controllers is that it can support both the traditional configuration and tuning of standard control idioms by nonprogrammers and the development of customized control algorithms by more experienced users. This is illustrated in the data flow diagram of Figure 3 which outlines the structure of a possible algorithm development package. Depending on the options chosen by the control engineer, control algorithms could either be configured from standard idioms as they are now on configurable controllers or developed in a higher level language (HLL). In either case, the user interface would verify the overall structure of the control algorithm against the system constraint table (e.g., flag specification of data table subscript range errors in custom algorithms or, equivalently, flag the specification of nonexistent input and output devices in an interactive configuration) and generate object code to be linked with a manufacturer-supplied utility library. This library would consist of algorithms to implement the standard control idioms and standard functions for the HLL compiler. The end

result of the control algorithm development process would be a programmable read-only memory (PROM) containing the executable machine code for the control algorithm. Controller reconfiguration would then be reduced to the process of changing PROM's. In fact, it would be useful to provide for multiple PROM's in the controller to facilitate reconfiguration or to allow alternative (e.g., more conservative) control strategies in the event of process upsets, grade changes or other contingencies that significantly alter the process characteristics from those for which the controller was designed.

To implement Chang and Seborg's algorithm in such a system, the control engineer would first determine how the sensors and actuators attached to the process were to be connected to the controller hardware, thus establishing where the measured variables $y_i(k)$, $i=1,2,\dots,m$ were stored in the measured variable table and where the manipulated variables $u_i(k)$, $i=1,2,\dots,r$ were stored in the control variable table. Set points for the state variables $x_i(k)$, $i=1,2,\dots,n$ (which might include some or all of the measured variables) would be assigned in the set point table and the constants w_i and M_i would be defined in the tuning parameter table. Finally, scratch pad variables would be allocated by index as either external (visible to the operator through the front panel) or internal (invisible to the operator). From there, implementation of Chang and Seborg's algorithm would be reduced to coding it in a higher level language, verifying the source code with the controller's development system, compiling and linking it with the necessary utility routines, and storing the result in a PROM to reside in the controller.

Summary and Conclusions

This paper has outlined the essential features of a hypothetical microprocessor-based controller that could be used to easily implement both traditional idiom-based and nontraditional model-based control system designs in an industrial environment. The controller structure represented in Figure 2 is not complete since it does not include certain important features that would be required in any practical controller, such as alarms and bumpless transfer between the automatic control mode(s) and manual operation. Similarly, the ability to remotely down-load new set points, tuning parameters or configurations is currently available on some configurable controllers and would be even more desirable in a highly flexible controller like the one described here. The main point, however, is that more flexible controllers are needed to facilitate the application of nontraditional control algorithms and the ideas presented here represent a basis for the development of such controllers. These controllers will require more extensive algorithm development software than standard configurable controllers do, but the advantages to both the end user and the manufacturer would be significant since new control idioms could be introduced as new revisions of the algorithm development package.

Finally, it is worth noting that first steps toward the development of such systems have been taken. For example, Technische Hochschule Darmstadt has developed a research controller specifically for the industrial application of nontraditional control algorithms⁹. This system is based on the Intel 8086 microprocessor with the AM 9511 floating point processor, 16K bytes of system software in PROM, 32K bytes of on-board RAM and 90K bytes of magnetic bubble memory. Also included are a real-time clock, eight differential input 12-bit A/D converters, and six 12-bit D/A converters. Control algorithms are developed in PL/M-86 on a host development system and down-loaded to the controller for execution. The system has been successfully employed to implement a two-input, two-output adaptive controller for a pilot air-conditioning plant⁹.

References

1. T. S. Chang and D. E. Seborg, "A Linear Programming Approach for Multivariable Feedback Control with Inequality Constraints," Int. J. Control, vol. 37, 1983, pp. 583-597.
2. E. Bristol, "Process Control: An Application Theorist's View of Control," IEEE Control System's Magazine, vol. 2, 1982, pp. 3-8.
3. M. Page-Jones, The Practical Guide to Structured Systems Design, Yourdon Press, New York, 1980.
4. H. Schuler, "Estimation of States in a Polymerization Reactor," Proc. 4th IFAC Conference on Instrumentation and Automation in the Paper, Rubber, Plastics and Polymerization Industries, Ghent Belgium, 1980, pp. 369-376.
5. C. E. Garcia and M. Morari, "Internal Model Control. 1. A Unifying Review and Some New Results," I&EC Process Design and Development, vol. 21, 1982, pp. 308-323.
6. P. Eykhoff, System Identification, John Wiley & Sons, New York, 1974.
7. D. G. Schultz and J. L. Melsa, State Functions and Linear Control Systems, McGraw-Hill, New York, 1967, chapter 4.
8. D. G. Luenberger, Introduction to Linear and Nonlinear Programming, Addison-Wesley, Reading, Mass., 1974.
9. F. Radke, "Implementation of Multivariable Parameter-adaptive Control Algorithms Using Modern Microprocessor Techniques," Proc. 6th IFAC Symposium on Identification and System Parameter Estimation, Arlington, VA., 1982, pp. 1466-1471.

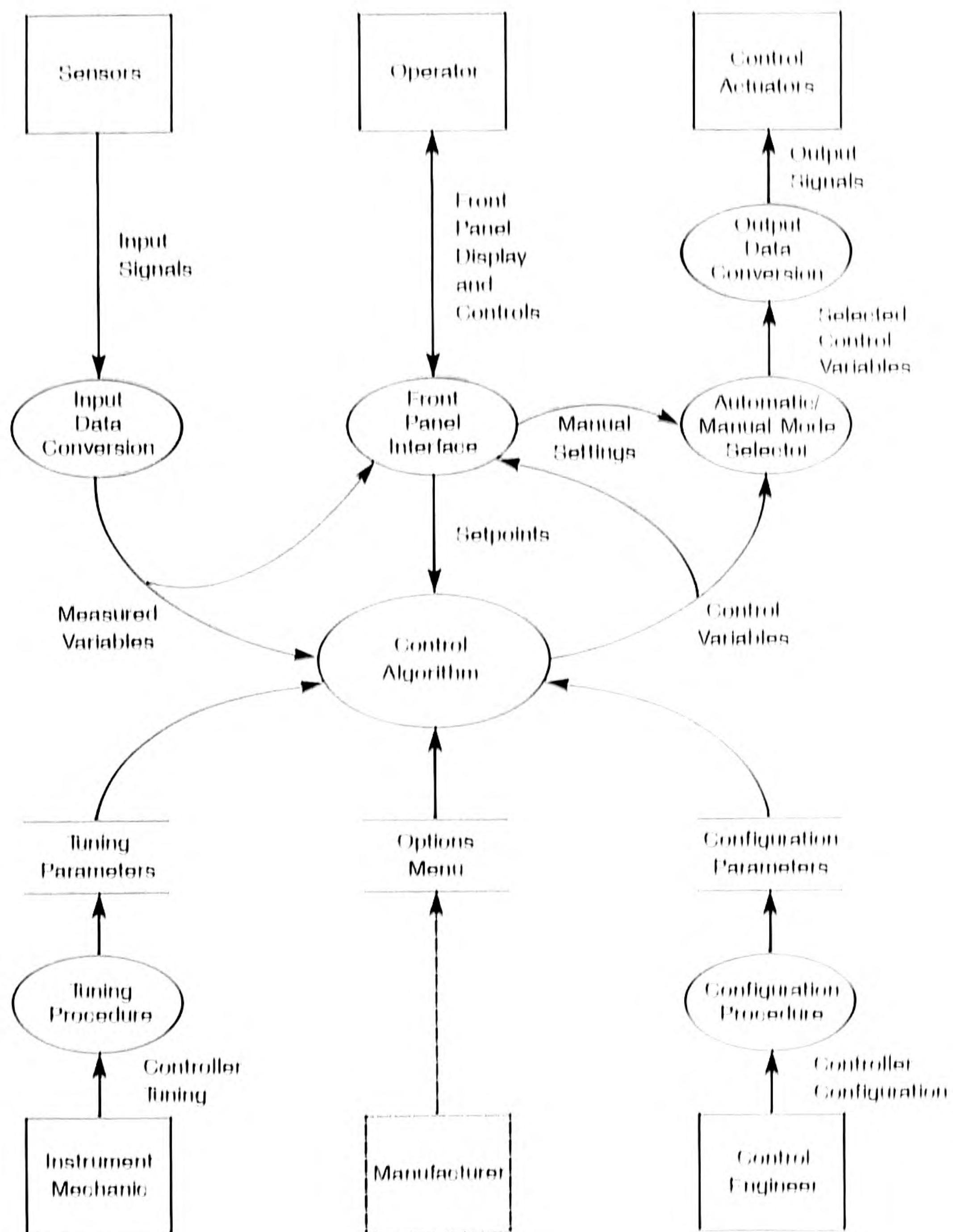


Figure 1. Traditional Configurable Controller

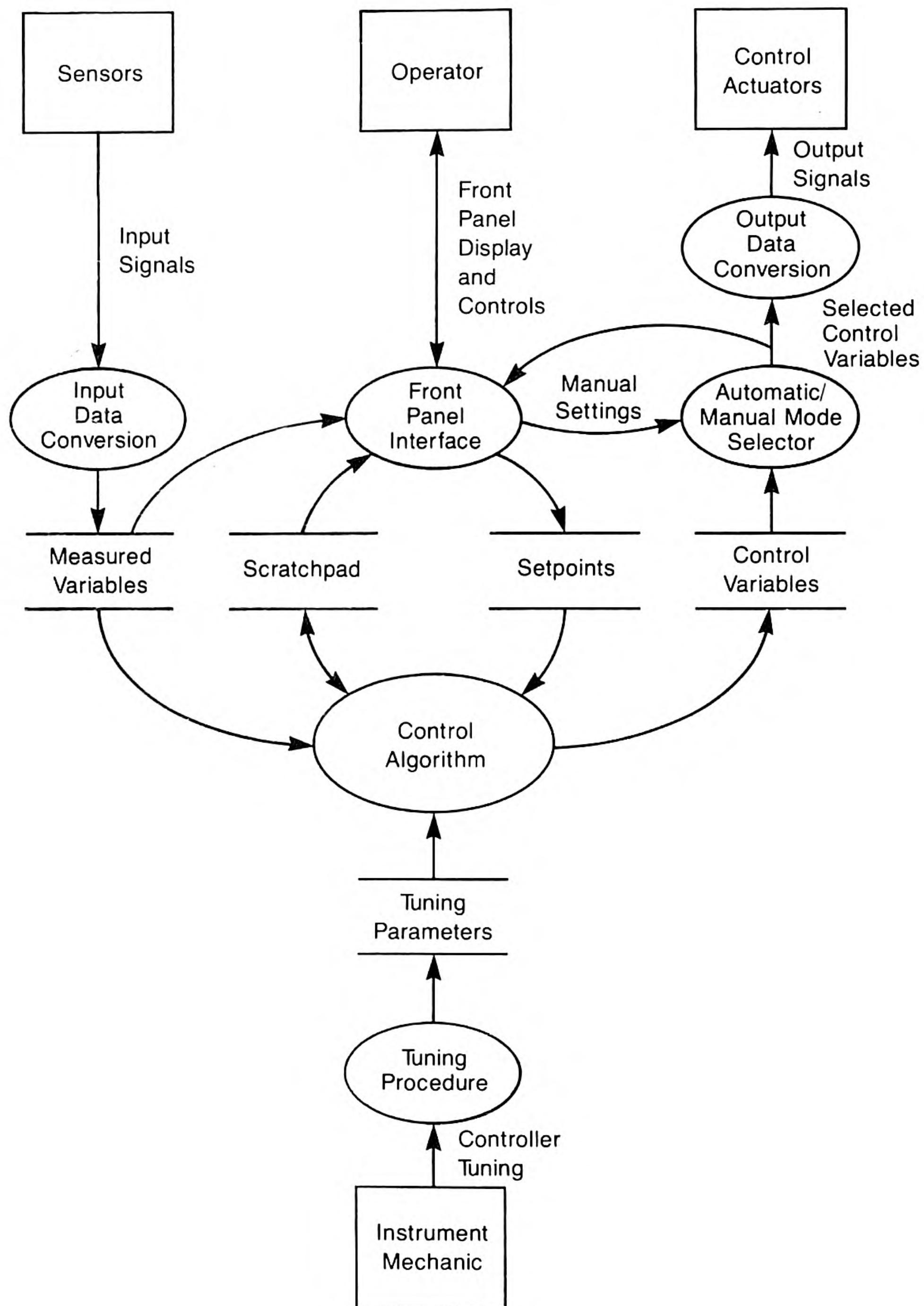


Figure 2. Hypothetical Flexible Controller

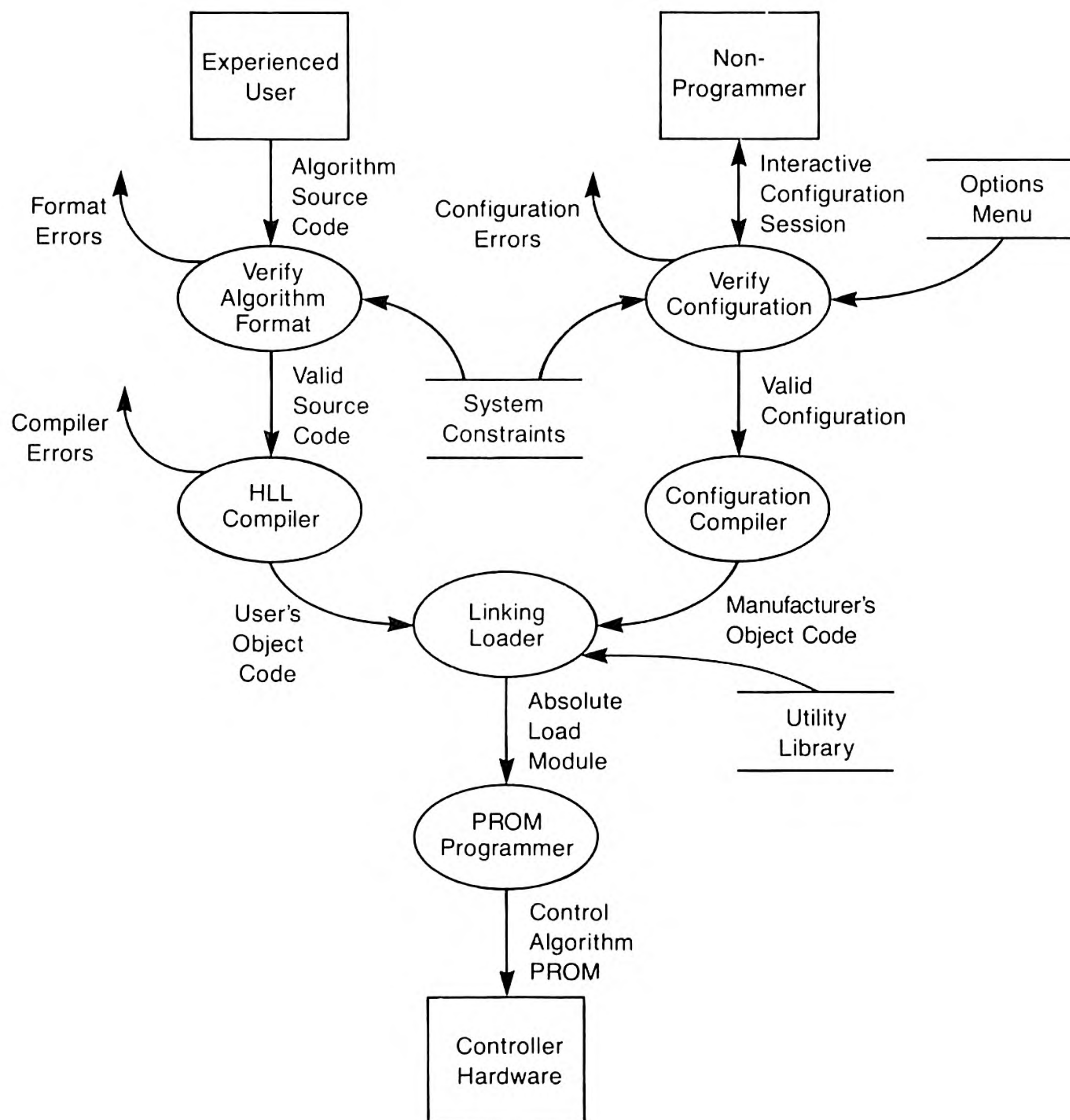


Figure 3. Control Algorithm Development

A STATIC MULTIVARIABLE FEEDFORWARD COMPENSATOR

by

MICHAEL J. PIOVOSO and RONALD K. PEARSON

ENGINEERING RESEARCH AND DEVELOPMENT DIVISION

ENGINEERING AND PHYSICS LABORATORY

ABSTRACT

Exact rejection of disturbances that occur in multivariable control systems is seldom possible. In this paper, a technique for the design of multivariable feedforward compensators for process control applications is presented. This technique is based on the notion of the generalized inverse and is applicable whether or not exact rejection is possible. Several design procedures are presented, one of which requires no a priori knowledge of either the system dynamics or of the other matrices in the state-equations. The technique is evaluated by designing a static multivariable feedforward controller and applying it to an evaporator.

INTRODUCTION

Although feedforward control has not been the active research area that feedback control has, single variable feedforward compensators have nevertheless been employed quite successfully in practice to improve the performance of systems operating under feedback control (Deshpande and Ash, 1981). In particular, if feedforward control can be used to reduce the impact of large upsets in the process inputs, feedback controllers can be tuned to give tighter control than would otherwise be possible, especially in highly nonlinear processes. The objective of this paper is to describe how feedback control can be improved by designing static multivariable feedforward compensators. These compensators are able to explicitly account for the influence of the disturbances on the process variables and manipulate variables. The feedforward controller allows for corrections for an arbitrary number of measurable disturbances with an arbitrary number of manipulated variables. Furthermore, the

design of the feed forward controller can be accomplished without the explicit knowledge of the system dynamics. The design requires the use of only simple step tests between inputs and outputs and disturbances and outputs, respectively.

Here, it is assumed that the dynamics of the process to be controlled can be described by either a continuous or a discrete time state-space model. Specifically, in the continuous time case, the process dynamics are described by

$$\dot{\underline{x}}(t) = A\underline{x}(t) + B\underline{u}(t) + L\underline{v}(t) \quad (1)$$

$$\underline{y}(t) = C\underline{x}(t) \quad (1a)$$

where $\underline{x}(t)$ is the n -vector of process variables describing its state at the time t , $\underline{u}(t)$ is an m -vector of manipulated variables available for control, $\underline{v}(t)$ is a p -vector of measurable disturbances and $\underline{y}(t)$ is the q -vector of process outputs. The matrices A , B , L and C are of dimension $n \times n$, $n \times m$, $n \times p$ and $q \times n$ respectively, and respectively represent the process dynamics, the manipulated variable influences on the state, the disturbance influences on the state, and the combination of state-variables which yield the process outputs. In the discrete time case, the process dynamics are described by

$$\underline{x}(k+1) = A\underline{x}(k) + B\underline{u}(k) + L\underline{v}(k) \quad (2)$$

$$\underline{y}(k) = C\underline{x}(k) \quad (2a)$$

where $\underline{x}(k)$ is the n -vector of process variables describing its state at the k th time step, $\underline{u}(k)$ is an m -vector of manipulated variables available for control, $\underline{v}(k)$ is a p -vector of measurable disturbances and $\underline{y}(k)$ is a q -vector of process outputs, as in the continuous time case. Similarly, the matrices A , B , L and C have the same dimensions and meanings as before.

Given the process model, note that if $m=n$ and B is nonsingular, it is possible to precisely cancel the effects of the disturbance $\underline{v}(t)$ on the process by applying the feedforward control

$$\underline{u}(t) = -B^{-1} L \underline{v}(t) \quad (3)$$

Such a choice of $\underline{u}(t)$ would exactly cancel the effects of the disturbances on the process outputs $\underline{y}(t)$ because the process outputs are linear combinations of the process states. Also note that unless $m=p$ and the feedforward gain matrix $B^{-1} L$ is diagonal, this control law cannot be implemented by pairing individual manipulated variables $\underline{u}(t)$ with individual disturbances $\underline{v}(t)$. Thus, designing single variable feedforward controllers for each pair is generally not possible using the control law of equation (3).

Furthermore, it is generally true that m is not equal to n so that B^{-1} cannot exist. It may still be possible to find a feedforward control $\underline{u}(t)$ that exactly cancels the effect of the disturbances, but only if the range of L is a subspace of the range of B . That is, there must exist some $m \times p$ matrix F such that $L = BF$, in which case the control law

$$\underline{u}(t) = -F \underline{v}(t) \quad (4)$$

will exactly cancel the effects of the disturbances on the states and the outputs of the process. Since this condition requires a very special relationship to exist between the measurable disturbance variables and the manipulated control variables, in general it cannot be satisfied. Consequently, it is desirable to determine how closely the control law of equation (3) can be approximated in cases where it cannot be implemented exactly. This leads naturally to the subject of generalized inverses (Ben Israel and Greville, 1980).

There are many generalized inverses, but the best known is the Moore-Penrose pseudoinverse (Ben Israel and Greville, 1980) which is defined for the $n \times m$ matrix B as the unique $m \times n$ matrix W which satisfies the following four conditions:

- i) $BWB = B$
- ii) $WBW = W$
- iii) $(BW)^* = BW$
- iv) $(WB)^* = WB$

where $(R)^*$ is the complex conjugate transpose of the matrix R . Note that if B^{-1} exists, it satisfies all of these conditions and since the Moore-Penrose pseudoinverse is unique, it reduces to the standard inverse whenever the latter exists.

It is also important to note that a unique Moore-Penrose pseudoinverse exists for any matrix B , regardless of its dimensions or rank. Replacing B^{-1} in equation (3) with the Moore-Penrose pseudoinverse W thus leads to the feedforward control law

$$\underline{u}(t) = -W L \underline{v}(t) \quad (5)$$

which can always be implemented. Physically, this control law represents the unique minimum-norm error, least-square solution of the exact disturbance rejection problem (Ben Israel and Greville, 1980). Stated another way, the driving energy for the homogeneous state-equation, $\dot{\underline{x}}(t) = A \underline{x}(t)$, due strictly to the disturbance, is minimal.

The control effort defined by equation (5) yields states which in turn get translated into disturbances in the process output. In general, this control law does not minimize the disturbances on the process outputs. An alternative approach would be to look at the effects of the disturbances in the steady-state outputs and define the control effort needed to minimize these disturbances. In steady-state, the state equations become

$$\dot{\underline{x}}(t) = 0 = A \underline{x}(t) + B \underline{u}(t) + L \underline{v}(t) \quad (6)$$

and

$$\underline{y}(t) = C \underline{x}(t) \quad (7)$$

Solving equation (6) for the states $\underline{x}(t)$ and substituting into equation (7) yields

$$\begin{aligned} \underline{y}(t) &= -CA^{-1} [B \underline{u}(t) + L \underline{v}(t)] \\ &= G \underline{u}(t) + H \underline{v}(t) \end{aligned} \quad (8)$$

Note that we have assumed that the inverse of A exists. In the open-loop case this implies that the determinant of $[sI-A]$ has no zeros at the origin. Stated somewhat differently, no pole of the open-loop transfer function exists at the origin. If the open-loop does contain a pole at the origin, a solution is still possible. By stabilizing the system with an appropriate controller the resulting closed-loop state equations would have an A matrix which was nonsingular; hence, the inverse would exist. Thus if the open-loop A is singular, stabilize the system so that the closed-loop A is nonsingular and then utilize that A . Also, note that matrices G and H are respectively the transfer functions from inputs to outputs and from disturbances to outputs evaluated at zero frequency. The matrices G and H have meaning only if the corresponding transfer functions are stable. However, as long as the system can be stabilized by some type of controller, the corresponding matrix A can be used to define the matrices G and H .

The control law derived from equation (8) is given by

$$\underline{u}(t) = -XHy(t) \quad (9)$$

where X is the generalized inverse of the matrix G . The solution given by equation (9) is the same as that of equation (5) if CA^{-1} is nonsingular or if the range of L is a subset of the range of B . In general, there are more states than outputs so that C is not square and hence CA^{-1} does not have an inverse. Also, note that the control law given by equation (9) yields a control effort which minimizes the integral squared error in the process outputs rather than in the states.

At first blush, the utility of this technique depends on a knowledge of the system matrix A or equivalently the dynamics of the system as well as the matrices B , C and L . But this is, in fact, not the case. One needs to know only the matrices G and H to utilize this technique. G and H are the DC gain matrices between inputs and outputs and disturbances and outputs respectively. Hence they can be determined by step tests. If one component of $\underline{u}(t)$, say u_k is set to a constant K , and the other inputs and outputs are held steady, the k th column of matrix G can be determined. This column is given by the changes in the components of the outputs $\underline{y}(t)$ normalized by the constant K . That is,

$$g_{jk} = y_j / K; \quad j=1,2,\dots,q \quad (10)$$

where g_{ij} is the ij th component of the matrix G , q is the total number of outputs, and y_j is the change in the j th component of $\underline{y}(t)$ due to the change in the i th component of the input $\underline{u}(t)$. By repeating this same procedure with the disturbances $\underline{v}(t)$, the matrix H can be determined. No knowledge of the system dynamics is required. The procedure involves utilizing simple step tests which are commonly performed in testing the performance of a controller.

To test these concepts, we have utilized the evaporator model of Bruun and Kummel [1979] and compare our results to those of Bruun and Kummel, who developed a dynamic feedforward compensator along with output-input pair matching to define their control strategy. Because of the character of the G matrix, it is possible, for this example, to exactly cancel the steady-state effect of the disturbances on the process outputs. However, it is not possible to cancel their effect on the states of the system.

EVAPORATOR MODEL

The evaporator to be modeled is shown in Figure (1). Table (1) defines the state, control, disturbance, and constant variables. Two supply tanks pump salt water into the first effect. Saturated steam is used to boil the solution. The bottom concentrates are pumped from effect one into effect two which is held under constant vacuum. The second effect is a climbing film unit. The steam from the first effect is used to concentrate product from the first effect bottoms into the second effect bottoms. The concentrated product flows from the bottom of effect two, and the steam is condensed and collected.

The model used in this paper is the same as shown in Table (1) for the evaporator. It is similar to other evaporator models developed by Bruun [1975]; Andersen, Glasson, and Less [1972]. The model consists of five states, three inputs, three outputs, and three disturbances. The model was generated from first principle mass and energy balance techniques. Bruun and Kummel operated the evaporator and compared the actual operational data with the model. The agreement between the two is quite good. For this paper, only simulation results are presented.

The control problem is primarily that of regulating the product concentration C_2 in effect 2. Disturbances enter into the feed flow F_0 and feed concentration C_0 . In addition, the solution holdups in effect one and effect two must be held within specified bounds. The process outputs are taken to be these two holdups and the product concentration C_2 . All three outputs are state variables.

RESULTS

Bruun and Kummel developed controllers by pairing controlled and control variables in a one-to-one fashion. The controllers were proportional gain only, and the parameters were established using frequency response and root locus techniques. The criterion of a damping ratio of at least 0.5 was applied. The closed-loop system was evaluated by observing the response to step disturbances in the input feed solution concentration and flow of plus or minus 25% and plus or minus 15% respectively. For purposes of this study, only the +15% change in the input feed solution is studied.

Four different pairings of the inputs and outputs were studied by Bruun and

Kummel. Table (2) defines the four pairings studied. In our work, we have used the control law defined by equation (5) to study all four pairings; that is, we have minimized the energy in the disturbances which drive the state-equations. In addition, we have utilized the control law defined by equation (9) to minimize the effect of the disturbances on the output.

Table (3) summarizes the effect of using feedforward via the control law of equation (5). Note that for each of the four pairings, one of the errors in the process outputs is driven to zero. This is purely coincidental and is not generally the case. Because we are minimizing the energy in the driving function for the state-equations, we expect some improvement in the offset errors due to the disturbances. In fact, the improvement varies significantly depending on the pairing chosen for control. Pairing 1 and feedforward yield total sum of square reductions of 71.5%, while pairing 4 yields only 32.2%.

The control law of equation (9) gives even better results. Since the inverse of the matrix G exists, the steady-state error in the process outputs can be driven to zero for each set of pairings defined in Table (1). We have verified the results for the first pairing only. Indeed, the steady-state errors are zero, but there is also a very significant reduction in the variations due to the transient behavior that was neglected. In fact, the performance of our static multivariable feedforward compensator out-performed the dynamic compensator proposed by Bruun and Kummel [1979]. Bruun and Kummel used the steam temperature to compensate the disturbances in input feed flow and composition. They designed their compensator to eliminate the steady-state error in the outlet composition. Hence, they had steady-state errors in the holdup in effects one and two of 16%. With the static compensator, there are no steady-state errors in any of the outputs.

Figure (2) illustrates the results for pairing 1 with and without feedforward compensation. Without compensation, both bottom concentrations C_1 and C_2 drop when the input feed flow increases 0.5 kg/min to 0.58 kg/min. When feedforward control defined by equation (5) is applied, some improvement is realized, but it is minimal. However, when static feedforward compensation designed to eliminate the steady-state effects on the process outputs is applied, the steady-state composition C_2 is the same as it was before the disturbance. But there is an effect on the composition C_1 . C_1 is not an output variable, and hence, there is no constraint on its steady-state behavior. Interestingly, C_1 has a smaller

steady-state error than it would have had with feedforward just aimed at the process states.

Figure (3) is another illustration of pairing 1 with and without feedforward compensation. The plots show the effects of feedforward on the holdup in effects one and two. In this case, feedforward on the states dramatically affects the holdups. The holdup in effect one, W_1 , is stabilized by the feedforward on the states, and the holdup in effect two, W_2 , has smaller steady-state error with feedforward than without. Since W_1 and W_2 are output variables, the feedforward aimed at the process outputs eliminates these steady-state errors.

Figure (4) shows the impact on the bottom flow for effects one and two. Since the holdup in effect one is controlled by the bottom flow of effect one and because the dynamics for the holdups are uncoupled, the steady-state error in holdup is proportional to the steady-state change in flow. The steady-state error in holdup for effect one is zero with both types of feedforward. Hence the steady-state change in the control effort F_1 is also zero. The steady-state error in effect two holdup is not zero for the case of feedforward via control equation (5). The corresponding flow F_2 also has a steady-state change for this case. However, since the holdup of effect two is an output variable, the procedure of equation (9) yields a zero steady-state error. As a result, the steady-state change in the corresponding flow is zero for feedforward aimed at the process outputs.

Finally, Figure (5) illustrates the effect of feedforward on the steam temperature and the temperature in effect one. Without feedforward, the steam temperature is forced to a much higher value. Feedforward to minimize the effect of the disturbances on the states results in slightly lower steam temperature and a temperature in effect one which is about the same as that achieved without feedforward. However, with feedforward to affect the process outputs, the steady-state change in steam temperature is very small. The temperature in effect one is now much higher.

SUMMARY AND CONCLUSIONS

Two different design approaches to generate a static feedforward multivariable compensator have been discussed. One, based on equation (5), chooses process inputs in a manner to minimize the effect of a disturbance in the sense that

the driving force for the homogeneous state-equations is of minimal total energy. This technique requires a knowledge of the matrices B and L . In general, this information is not readily available, and acquiring this knowledge may in itself involve as much effort as modeling the entire dynamics of the process. Furthermore, not all the states may be measurable.

The second procedure is a static compensator which is designed to minimize the effect of disturbances on the process outputs. Such a design requires two matrices G and H defined by equation (8). G and H are the DC gain matrices for the inputs to outputs and disturbances to outputs, respectively. Equation (10) defines the relationship between the step response and the G matrix. Both matrices, G and H , are thus determined by simple step tests on the

process inputs and disturbances. Having found G and H , the compensator function is defined by equation (9).

These concepts were tested on a model provided by Bruun and Kummel [1979]. Simulation results indicate that significant improvement is possible by feeding forward the disturbance according to equation (5). However, much better results are possible by choosing the control inputs in a manner to minimize the steady-state effects on the process outputs. Furthermore, the design of the compensator is straight forward and involves no knowledge of the process dynamics. Simulation results indicate that significant improvement is possible over even dynamic compensation and without the design problems associated with a compensation scheme.

Table 1

Fifth Order Evaporator Model Notation,
Associated Steady State Operating Point and State Space Model

Nomenclature

State Variables

W_1 solution holdup effect 1
 C_1 solution concentration
 effect 1
 T_1 temperature inside effect 1
 W_2 solution holdup in effect 2
 C_2 solution concentration
 effect 2

Disturbance Variables

F_0 feed solution flow
 C_0 feed solution concentration

Control Variables

F_1 bottom flow effect 1
 F_2 bottom flow effect 2
 T_s temperature of supply
 steam in coil

Constant variables

T_a ambient temperature
 T_2 temperature inside effect 2

Table 1 (continued)

Steady state operating conditions for state space model

W_1 --8.0 kg F_1 --0.33kg/min F_0 --0.5 kg/min T_a --20°C
 C_1 --3.05 g/kg F_2 --0.18 kg/min C_0 --2.0 g/kg T_2 --60°C
 T_1 --107°C T_s --120°C T_0 --16°C
 W_2 --3.5 kg
 C_2 --5.62 g/kg

State space model

$$\begin{bmatrix} W_1 \\ C_1 \\ T_1 \\ W_2 \\ C_2 \end{bmatrix} = \begin{bmatrix} 0.0 & 0.0 & -0.0034 & 0.0 & 0.0 \\ 0.0 & -0.041 & 0.0013 & 0.0 & 0.0 \\ 0.0 & 0.0 & -1.1471 & 0.0 & 0.0 \\ 0.0 & 0.0 & -0.0036 & 0.0 & 0.0 \\ 0.0 & 0.094 & 0.0057 & 0.0 & -0.051 \end{bmatrix} \begin{bmatrix} W_1 \\ C_1 \\ T_1 \\ W_2 \\ C_2 \end{bmatrix} +$$

$$\begin{bmatrix} -1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.948 \\ 0.916 & -1.0 & 0.0 \\ -0.598 & 0.0 & 0.0 \end{bmatrix} \begin{bmatrix} F_1 \\ F_2 \\ T_s \end{bmatrix} + \begin{bmatrix} 0.0 & 0.0 & 1.0 \\ 0.062 & 0.0 & -0.132 \\ 0.0 & 0.039 & -7.189 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix} \begin{bmatrix} C_0 \\ T_0 \\ F_0 \end{bmatrix}$$

Table 2

Configuration number and associated pairings of
controlled and control variables

1	2	3	4
W_1-F_1	W_1-T_S	W_1-F_1	W_1-T_S
W_2-F_2	W_2-F_1	W_2-T_S	W_2-F_2
C_2-T_S	C_2-F_2	C_2-F_2	C_2-F_1

Table 3

Control Method	Controller Definition	Poles	Offsets to Disturbance of 15% of F_0		
			Without Feedforward	With Feedforward of Equation (5)	Reduction in Sum of Squared Errors
Multiloop	$F_1 = 0.2 W_1$	- 0.055 - 0.1	$W_1 = 0.30$	$W_1 = 0.0$	
Config. 1	$F_2 = 0.1 W_2$	- 0.229+j0.152	$W_2 = 0.34$	$W_2 = -0.20$	71.5%
Prop.	$T_S = -41 C_2$	- 0.925	$C_2 = -0.19$	$C_2 = -0.17$	
Multiloop	$F_1 = 0.3 W_2$	- 0.08	$W_1 = 0.78$	$W_1 = 0.63$	
Config. 2	$F_2 = 0.21 C_2$	- 0.10+j0.16 - 0.11	$W_2 = -0.18$	$W_2 = 0.0$	37.6%
Prop.	$T_S = 12.0 W_1$	- 1.1	$C_2 = 0.12$	$C_2 = 0.11$	
Multiloop	$F_1 = 0.2 W_1$	- 0.055	$W_1 = 0.28$	$W_1 = 0.0$	
Config. 3	$F_2 = 0.21 C_2$	- 0.038+j0.2 - 0.20	$W_2 = 0.39$	$W_2 = 0.31$	55.8%
Prop.	$T_S = 24.0 W_2$	- 1.11	$C_2 = 0.12$	$C_2 = 0.11$	
Multiloop	$F_1 = 0.27 C_1$	- 0.06 - 0.10	$W_1 = 0.47$	$W_1 = 0.39$	
Config. 4	$F_2 = 0.1 W_2$	- 0.135+j0.14	$W_2 = 0.23$	$W_2 = 0.24$	32.2%
Prop.	$T_S = 21.0 W_1$	- 1.12	$C_2 = 0.20$	$C_2 = 0.0$	

BIBLIOGRAPHY

Andersen, Glasson and Less 1961 -

J. A. Andersen, L. W. A. Glasson, and F. P. Less, "The Control of a Single-Effect Concentrating Evaporator," Trans. Soc. Inst. Tech. 13 (1), 21.

Ben Israel and Greville 1980 -

A. Ben Israel and T. N. E. Greville, Generalized Inverses: Theory and Applications, Robert E. Krieger Publishing Co., Huntington, N.Y. 1980.

Bruun 1975 -

N. G. Bruun, Multivariable Computer Control of a Double Effect Pilot Plant Evaporator, Ph.D. Thesis, Dept. of Chemical Engineering, Technical University of Denmark.

Bruun and Kummel 1979 -

N. G. Bruun and M. Kummel, "Multiloop, Feedforward, Modal and Optimal Control of an Evaporator," Automatica 15, 269.

Deshpande and Ash 1981 -

P. B. Deshpande and R. H. Ash, Elements of Computer Process Control with Advance Control Applications, Instrument Society of America, 1981.

Newell and Fisher 1972 -

R. B. Newell and D. G. Fisher, "Experimental evaluation of Optimal Multivariable Regulatory Controllers with Model-Following Capabilities," Automatica 8, 247.

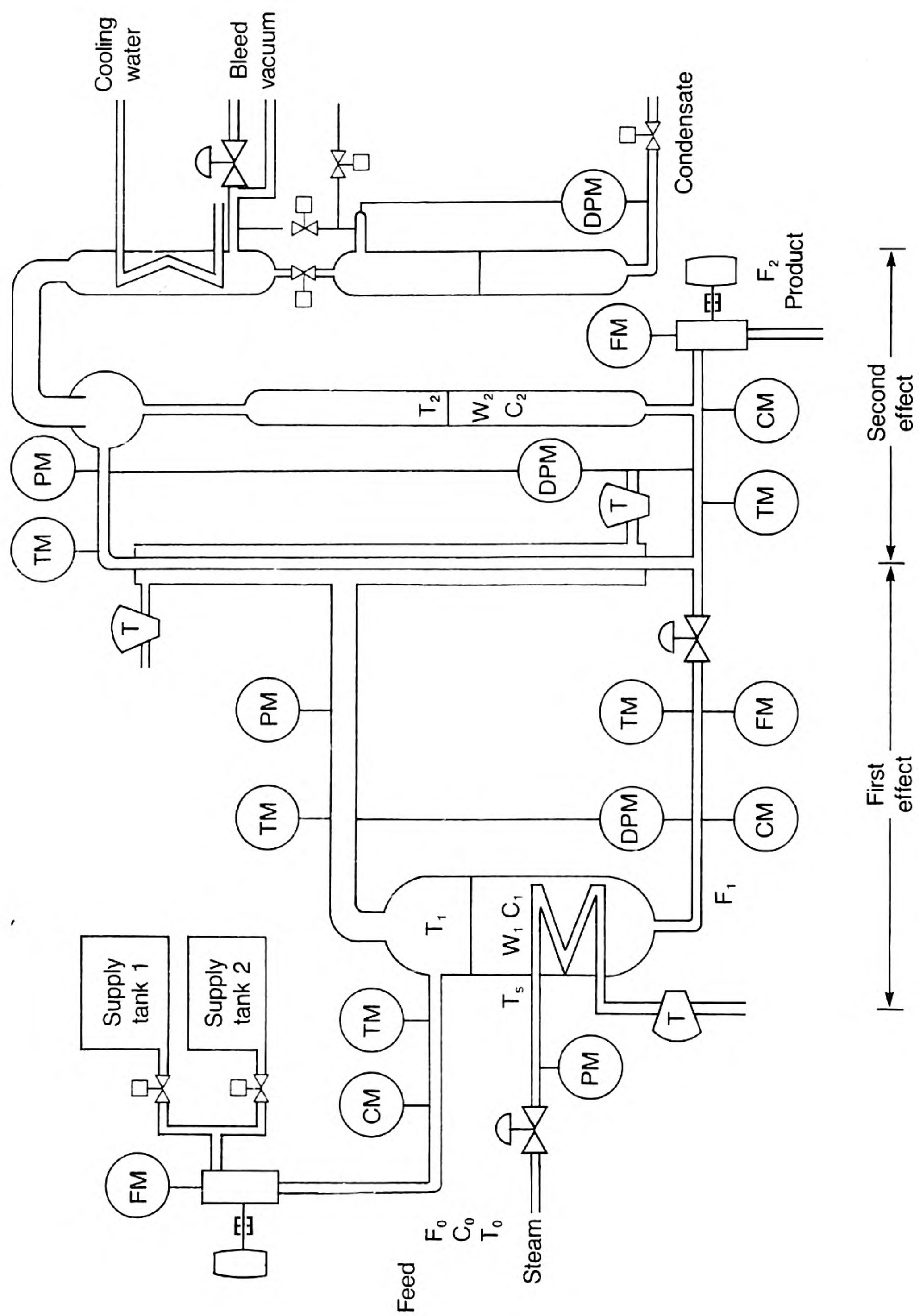


Figure 1

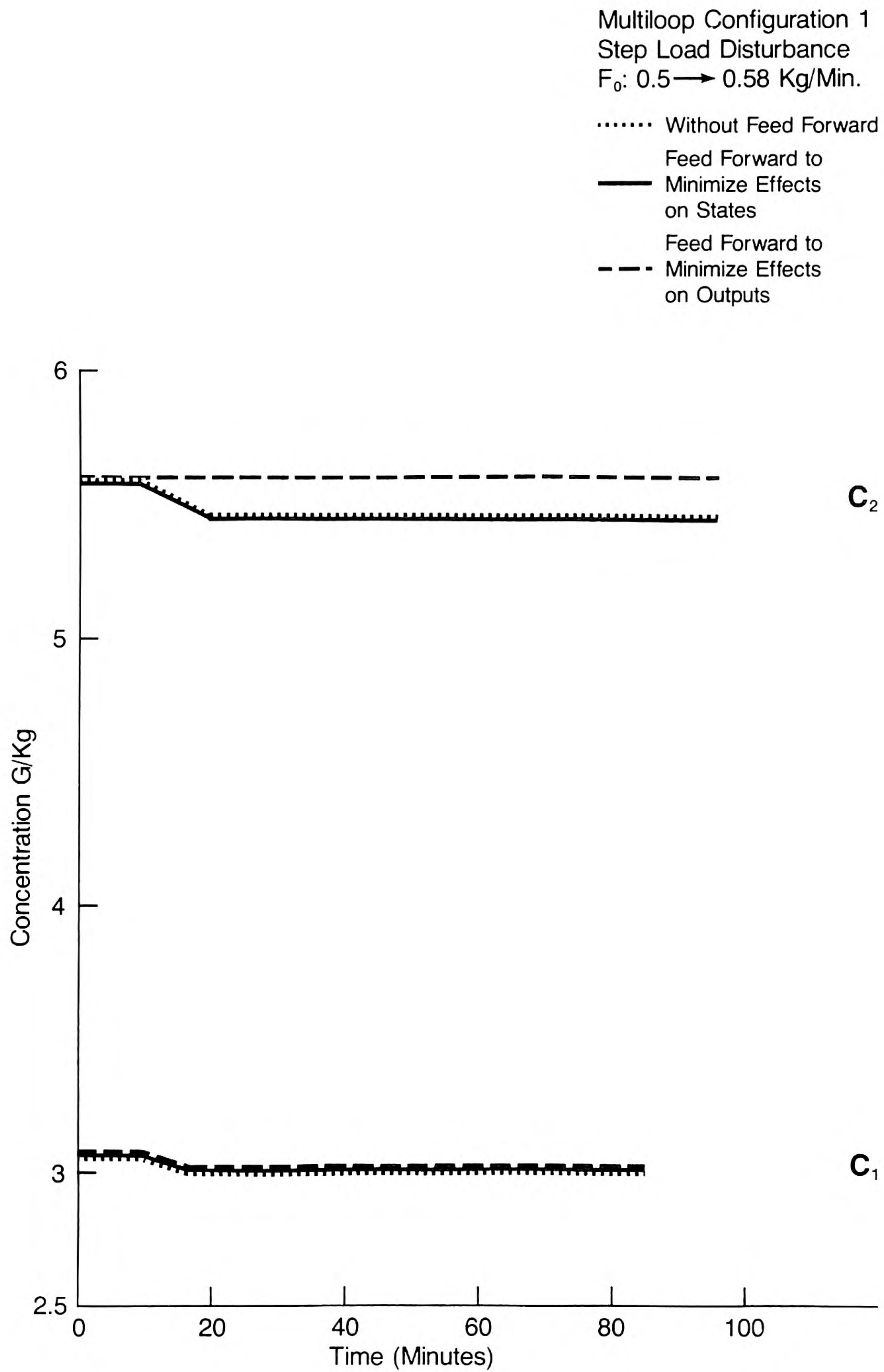


Figure 2

Multiloop Configuration 1
 Step Load Disturbance
 $F_0: 0.5 \rightarrow 0.58 \text{ Kg/Min.}$

..... Without Feed Forward
 — Feed Forward to Minimize Effects on States
 - - - Feed Forward to Minimize Effects on Outputs

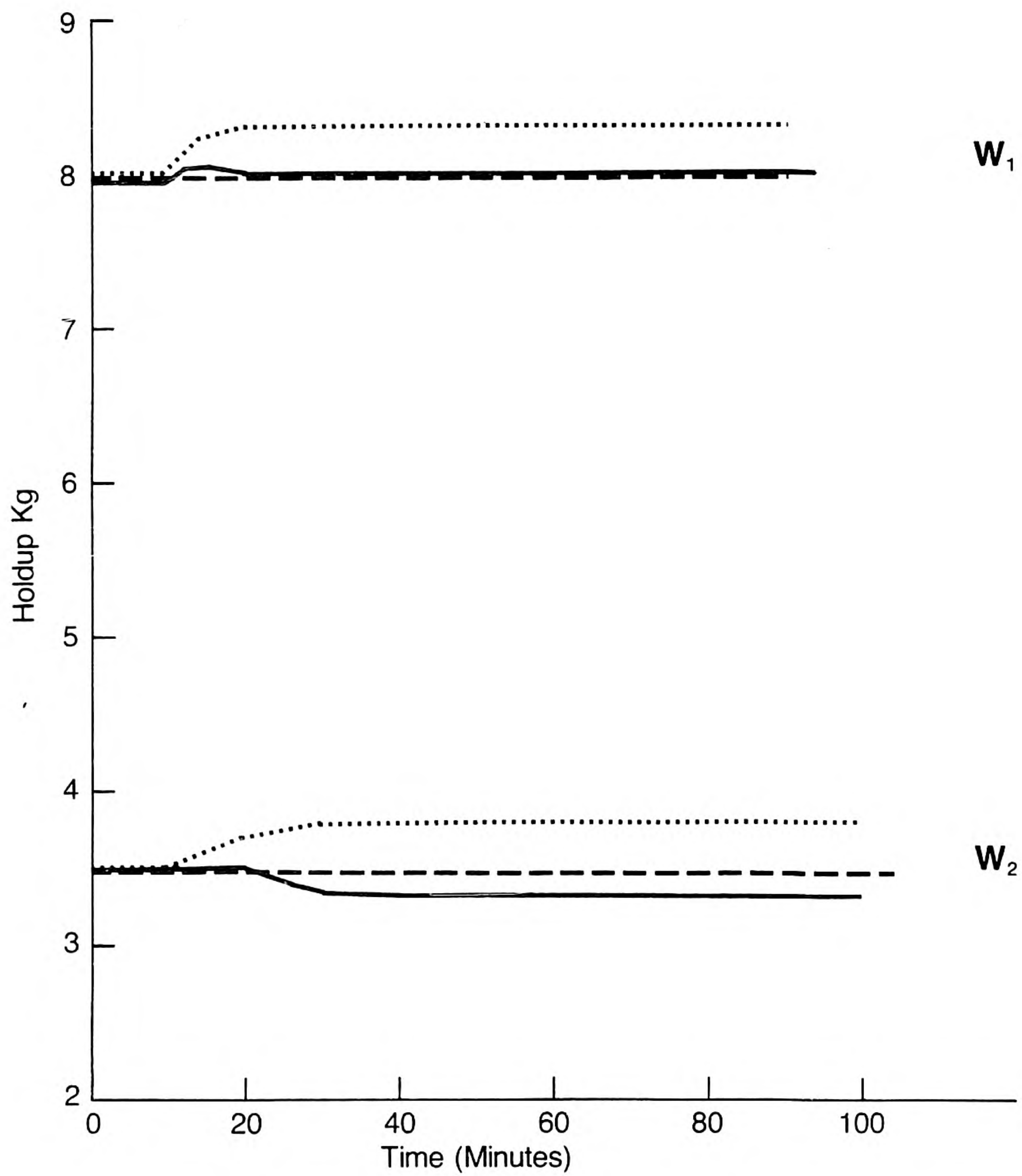


Figure 3

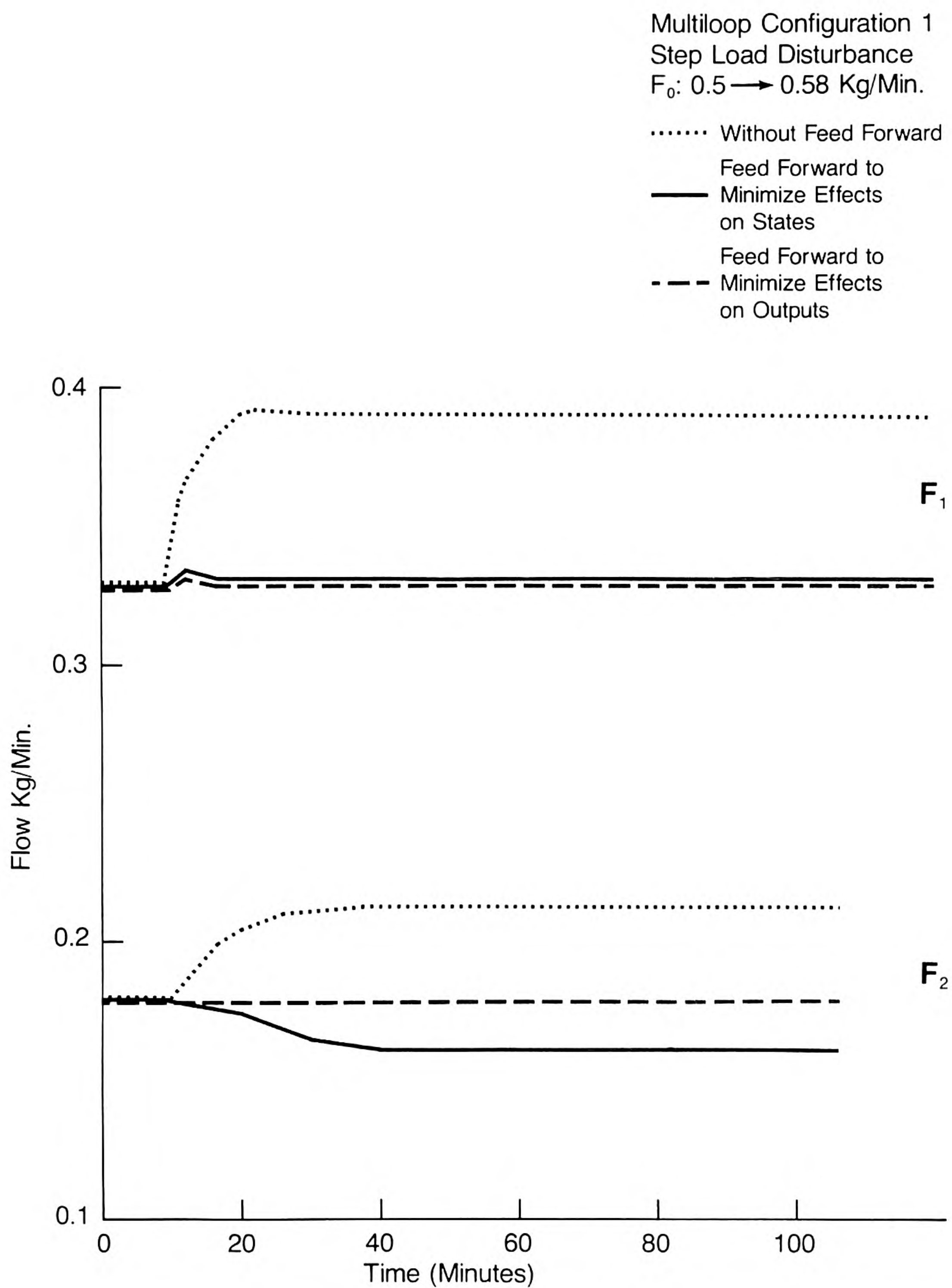


Figure 4

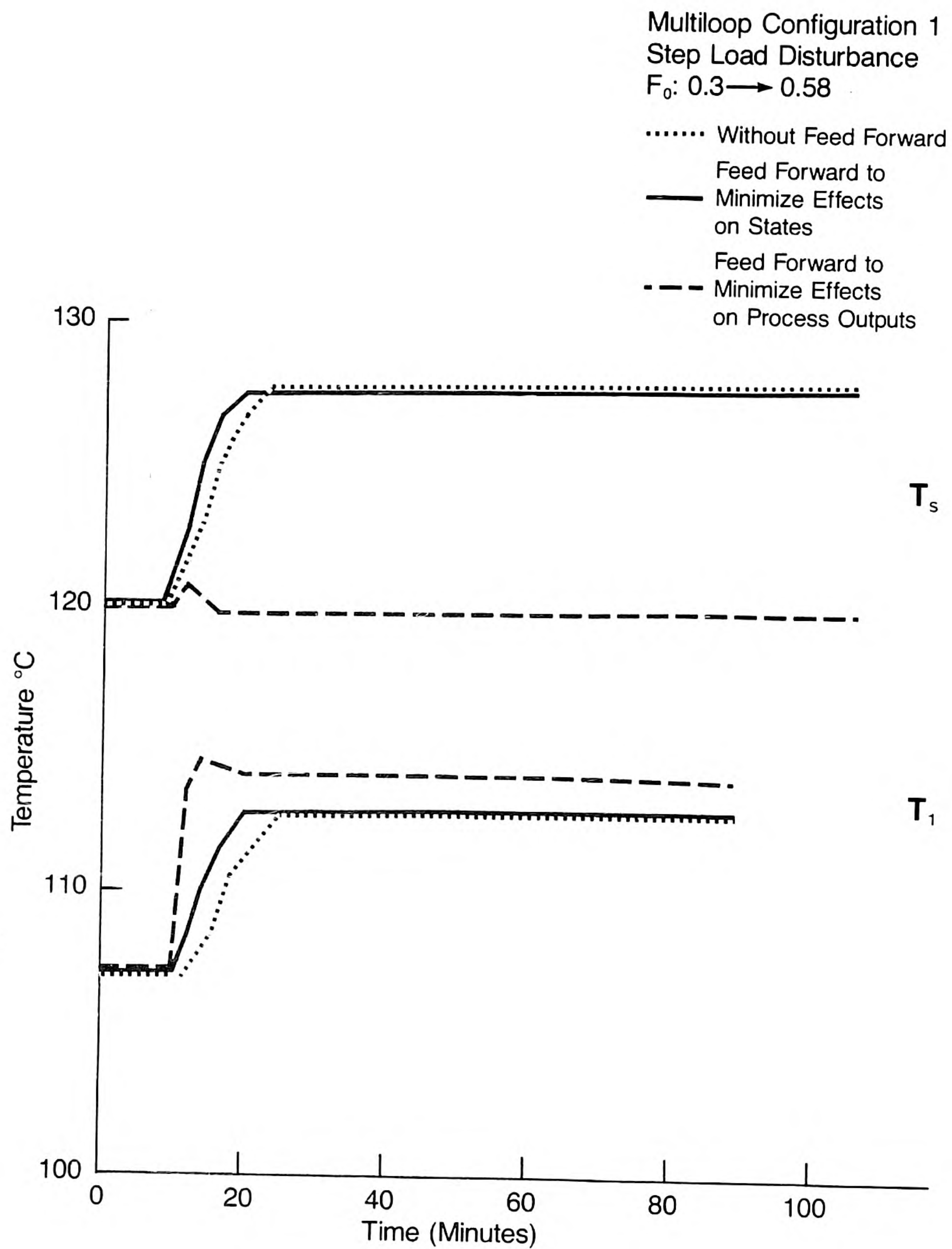


Figure 5

INUZ - A PROJECT FOR INTERCONNECTING COMPUTER NETWORKS

Zhigang Fan*

Institute for Informatics
University of Zürich
Zürich, Switzerland

Abstract -- INUZ (InterNetwork of the University of Zürich) is a project to connect the heterogeneous local networks at the University of Zürich. This paper intends to give an overview of the INUZ project. Its topology, protocol architecture, ITP-CM protocol, gateway design, implementation and performance are described.

I. Introduction

The more computer networks, especially local area networks come into use, the problem of interconnecting them also draws more attention. Whereas the first articles on internetworking appeared in the mid 1970's [1,2]. It is only now, that a larger clientele is being confronted with this subject.

INUZ (InterNetwork of the University of Zürich) is a project to connect the local networks at the University of Zürich and thus provide the users with services such as file transfer, electronic mail, remote login, interprocess communication, etc. over several networks.

The university consists of more than 60 institutes. In the nearest future, it is expected that most institutes will have several 16-bit microcomputers with Winchester disks. Some institutes will have their own local area computer networks. It would certainly be easiest for a university to evaluate a standard local area network for all their users. The problem of interconnecting them would be easier to handle due to the homogeneity of the network.

However for several reasons we do not believe that such an approach would be realistic:

1. The demands on local area networks may differ according to application. For example, a local network in an office environment may not need as high a data rate as a network connecting laboratory equipment, or as a network used for process control.
2. As local area networks become more popular, computing products appear on the market of which a specific local area network may be a part. For example, a workstation system could include a specific local area network to link several workstations and a disk server.
3. Local area networks evolve. New products appear on the market. These new products should be able to be integrated into an existing environment.

Therefore, the local networks we are going to connect will be heterogeneous in nature.

In the following sections, the design considerations are presented. Some aspects, such as communication architecture and protocol design, are described in more detail in [3].

II. Topology

With more than 100 buildings scattered throughout Zürich, University of Zürich is geographically quite distributed. Topologically, there are two possible approaches in interconnecting the networks [4].

One is to interconnect local networks in a mesh way, i.e. every local network can link with others as will. This would lead to an intricate structure and complicated routing schemes.

* Visiting Scholar from the People's Republic of China.
Permanent address: Shanghai Teacher's College, Shanghai.

Another solution, and it is what we have adopted, is to have a hierarchical topology. At the top layer, a network, which serves as a backbone for internetwork communication, connects all the local networks. The lower layer consists of different local networks, which connect diverse user devices. Every local network will link to the backbone through a gateway. Figure 1 demonstrates this hierarchical topology.

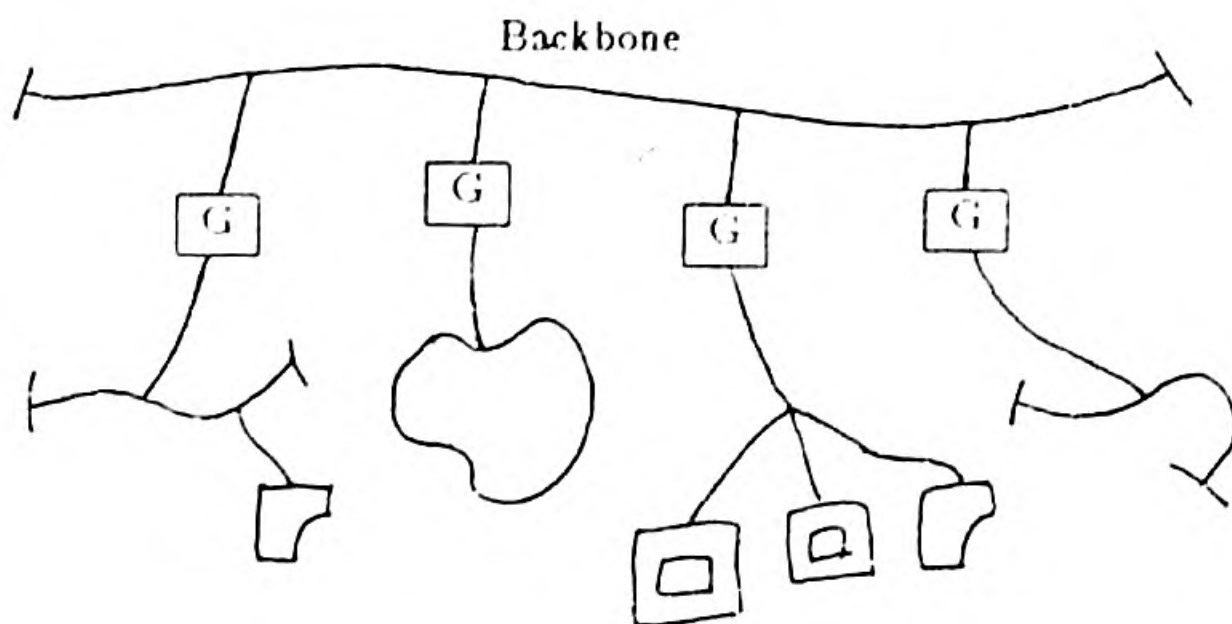


Figure 1.

The backbone network should be able to extend over large distances. We chose a broadband local area network for our backbone. A network using this technology can link the users over larger distance than a baseband network can. It is able to transmit data for 25 or more kilometers. Theoretically a long-haul or a privately owned X.25 network could also be used as backbone. But a broadband local area network leads to a simpler structure. It has a tree-like topology, so that all of the geographically distributed buildings could easily be reached.

One of the advantages of this hierarchical topology is that it leads to a straightforward routing algorithm and thus also makes the gateway design simpler. Every internet packet goes through the local gateway to the backbone, and via a remote gateway to the destination host. As we also adopted a hierarchical address scheme, which has an explicit network address field, it is quite easy for the local gateway to route the internet packet to the remote gateway.

III. ITP-CM Protocol

The architecture and protocol design of INUZ were

strongly influenced by several factors:

1. The local networks we are going to connect will considerably differ in technology, architecture and performance.
2. Most of our local networks are expected to have their own local transport services, which provide virtual circuits between users on the same network. Many of the networks will not offer datagram services.
3. The end users of INUZ include many small microprocessors. Therefore the protocols should be simple enough to be implemented in hosts with limit resources.

To account for point 1 above, we adopted an "internetwork protocol" approach instead a "relay" approach [5,6]. In the relay approach, network interconnection is done by relaying a series of local networks. No common protocols are required above the local services. At gateways, the protocols of one network must be translated into those of the next network. As we know, protocol conversion between two different networks is generally very complicated. For example, one network could offer virtual circuit service, while the other uses datagrams. The addressing scheme could be totally different.

When using an "internetwork protocol" approach, however, no translation is needed at the gateways. A common internetwork layer is used to transmit global information. Because of the diversity of the networks we are coping with, The internetwork protocol approach is the only realistic approach.

Nearly all of the popular standards, which use the internetwork protocol approach, base their internetwork protocol on connectionless local network services. For example, the Department of Defence TCP/IP model [7,8] uses the Internet Protocol (IP) as the lowest internetwork protocol. The IP transfers internet datagrams over the different networks and provides a base for the Transmission Control Protocol (TCP), which offers internet end-to-end services. Because IP transmits internet datagrams, it fits much nicer into environments which use networks supporting datagrams. Because of point 2 mentioned above, we feel that we cannot use internetwork models of this type. (Otherwise a virtual circuit would have to be established separately for every datagram.) We thus saw it necessary to design and

develop our own internetwork model and associated protocols.

The model designed for INUZ has four protocol layers, as shown in Figure 2. The lowest layer consists of local networks with their own transport services. For networks which do not offer virtual circuits, an additional local transport layer should be added to offer this local service. The shaded area of Figure 2 represents this additional layer. A Circuit Managing (CM) layer is the second layer. Its function is to route internetwork packets and to manage local network transport virtual circuits by opening and closing them. It concatenates a chain of local network virtual circuits to connect the internetwork end users. An Internet Transport Protocol (ITP) was developed to reside above the CM layer. This protocol offers an end-to-end internetwork transport service. The top layer is, as usual, the Application layer.

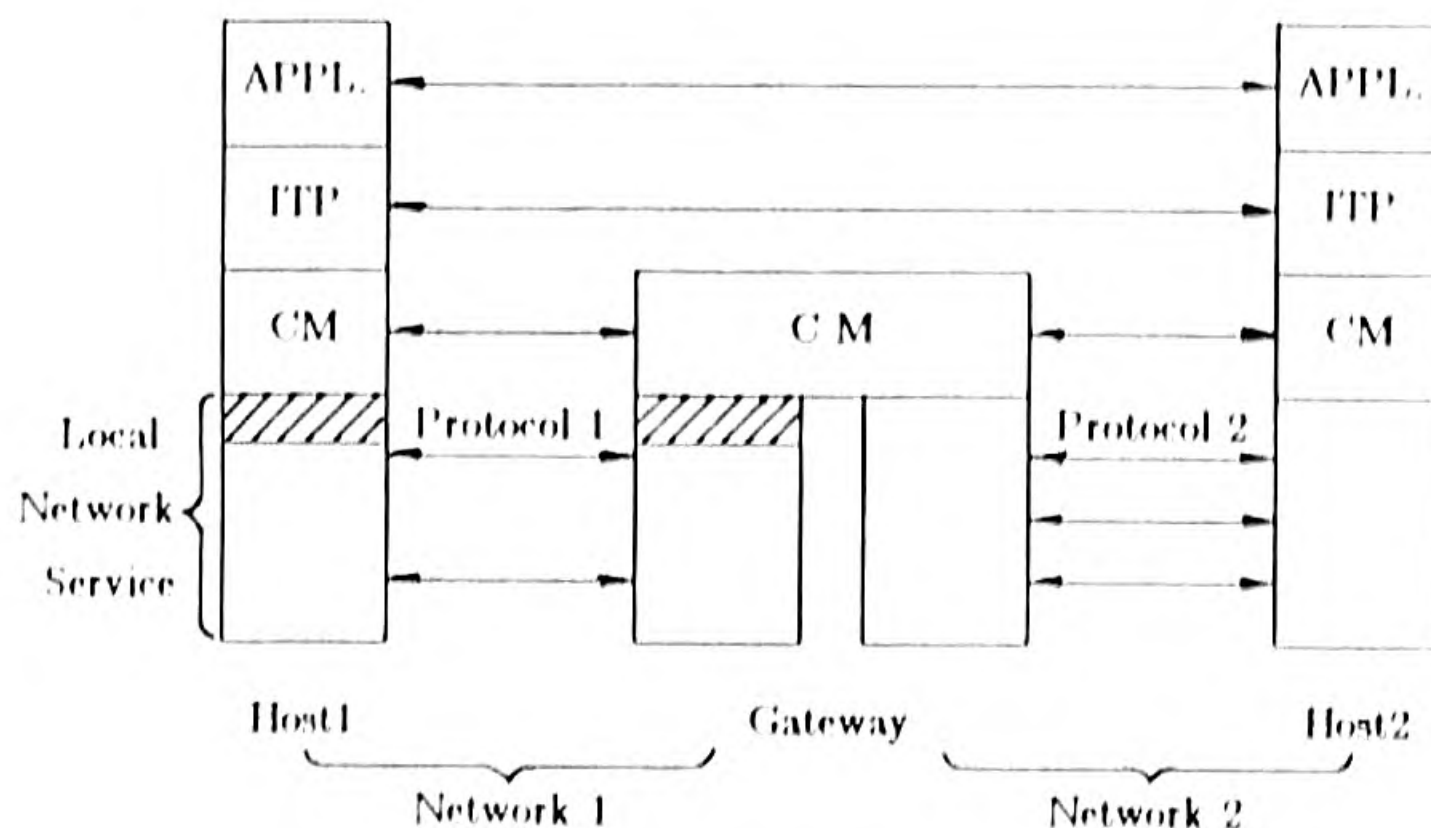


Figure 2.

The local network transport layers provide error-free and flow-controlled virtual circuits between hosts and gateways and between two gateways. The CM layer concatenates a series of such virtual circuits (at most 3 because of the chosen topology) as a service for the ITP layer. The services CM offers, although not end-to-end, are quite reliable. Misorder is nearly impossible. Bit errors and loss of packets can only happen when a memory error in a gateway occurs. This was exploited while designing the ITP-protocol and resulted in a simpler and more efficient protocol.

The ITP protocol has 6 main packet types: CallReq, CallAck, Data, Nack, Poll and Close. An ITP connection

is established after a negotiation between two ITP parties, in which CallReq and CallAck packets were exchanged. Connections are disconnected after Clear packets have been exchanged. Nack packets are used to ask for a retransmission.

We have designed an improved variable-size sliding window mechanism for end-to-end flow control. It differs from the ordinary ones especially in the technique used for detecting missing packets and crashed hosts.

Because misorder of packets in the CM layer is nearly impossible, the receiver can accept packets strictly according to the Sequence Number one by one. If the Sequence Number is greater than expected, (which most probably implies the loss of one or more packets), it sends a Nack packet, asking for retransmission.

When sending the last packet in the window or the last packet of a higher level message, a Poll bit in the packet is set to request that the partner return an acknowledgement.

Only in some critical circumstances, for example, when the sender is waiting for a new window in order to be able to transmit more packets, will a timer be started. If a timeout should occur, ITP will send a Poll packet to request for an acknowledgement.

The objective of this strategy is:

1. To detect the crash of a gateway or of the partner host, and to prevent the user from hanging on a "half-closed" connection.
2. To prevent deadlock caused by an occasional loss of packets which had the Poll bit set, or caused by lost acknowledgements.

The advantage of this error and flow control mechanism is that, it requires less resources. In ordinary sliding window mechanisms, the receiver buffers all in-coming packets which are out of order, but still within the window. In ITP, the receiver does not have to do so.

In ITP, the sender only starts the timer in some special circumstances. Compared with some other sliding window schemes, where a timer is started for every outstanding packet, ITP needs only one timer per connection.

ITP also optionally provides the possibility of not using the end-to-end mechanism. Because of ITP's quite reliable underground, users who have no special

demands on reliability, but rather prefer a higher throughput, better response time and lower buffer space expenses, can relinquish the end-to-end flow and error control service. They can decide not to use error control or both error and flow control when establishing the connection.

IV. Gateway

The topology and communication architecture adopted by INUZ leads to a simple gateway design.

The gateway linking a local network and the backbone network could be a special-purpose gateway machine, or one of the hosts on the local network. The hardware of our gateway is based on the Multibus and Motorola 68000 processor. Two interfaces are needed. One standard interface will always connect to the backbone. The interfaces to the individual local networks are network dependent. This is shown in Figure 3.

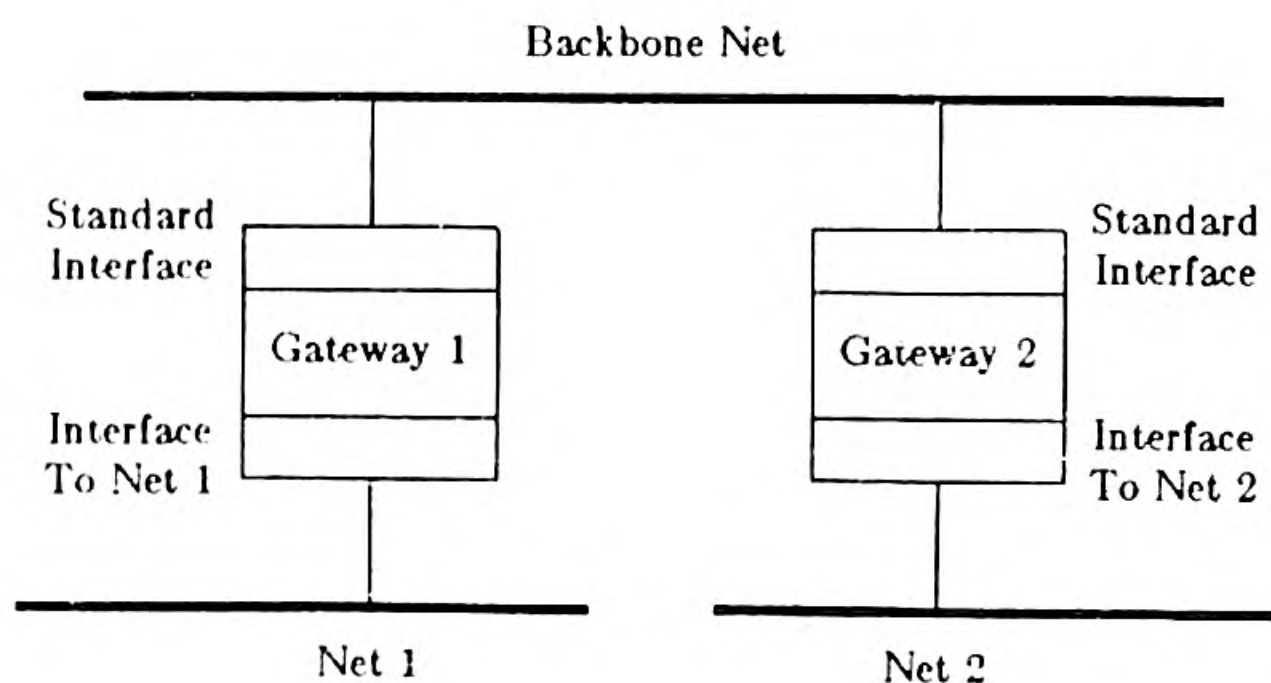


Figure 3.

Because of the layered architecture, gateways do not have to do any protocol conversion. The software in the gateways covers two layers of our 4-layer model. The higher of the two is the CM layer. Its main functions are:

1. When users from the application layer try to establish an ITP connection, it should help concatenate local virtual circuits to link the end users.
2. After an end-to-end connection has been established, it just passes the ITP packets transparently from one local network to the other.

3. When the ITP connection is closed, it should close the local virtual circuits which are not needed any more.

The layer below CM consists of two parts. One part implements the local transport layer for the backbone network, whereas the other part implements it for the other connected local area network.

It is thus necessary to adapt only a small portion of the gateway software to take the heterogeneous local networks into account.

V. Implementation and Performance

We have implemented ITP-CM protocol on several different systems, including PDP-11, Onyx and Sun workstation systems [9].

Onyx is a Zilog Z-8000 based microcomputer with the Unix V7 multiuser operating system. The Sun workstation is a personal workstation based on Motorola 68000 processor. It uses the Unix BSD4.2 operating system.

The Sun workstations are all connected to an 10 mbps Ethernet local network, whereas many of the other systems are connected to the LocalNet-20 [10] network developed by Sytek Corporation. This is a broadband local area network, where the devices are connected to front-ends, called Tboxes, via RS-232 ports. These front-ends were developed, by Sytek, mainly to support terminal-computer communications. The only services offered to end users are commands to manage sessions over the network. After a session between two ports has been established, every byte input is transmitted transparently to the second port. Six layers of protocols are implemented in the front-end to achieve this. (The claim is made that the architecture adheres to the OSI model.) Although we do not necessarily need all these layers, it is not possible to relinquish several of them, due to the proprietary nature of the network.

The communication protocols can be implemented either in a centralized way or in a decentralized way. In the first case, the communication software resides in processes which run in the background. We will call these processes collectively *transport station*. In the second case, communication software is kept in library routines, which can optionally be linked as user program. We chose the first approach for nearly all our systems, because a centralized transport station can

better manage resource distribution, statistic compile, etc. However, in some special situations, for example, in single-process systems, the second approach could be appropriate, for its simple implementation.

A major restriction we imposed upon ourselves was that no changes were to be made to the operating system. This was necessary due to the fact that source licences were not available for all our systems.

User processes communicate with the transport station through some local interprocess communication mechanisms. On systems where this is not possible, such a mechanism must first be developed. In Unix V7, nonrelated processes cannot communicate with each other locally. Thus a means for local interprocess communication had to be developed first[11].

On the systems connecting to LocalNet-20, the transport station mainly consists of two standby processes linked by a pipe. One of them implements the ITP and CM layer, whereas the another process implements local transport protocol.

The layer for the local transport protocol is needed, even if the system is connected to the LocalNet-20 network which offers virtual circuits, as the reason being that the virtual circuits are not truly end-to-end. The communication protocols are implemented in the Tboxes. Errors can and do occur when passing data between the host and Tbox. So this additional layer is used to detect and correct these kind of errors, and thus makes the virtual circuits within one local network truly end-to-end.

The CM layer is composed of several procedures, which maintain a table of the information of local virtual circuits. The table contains such items as the destination, the status of a virtual circuit and the number of ITP connections pass through it.

In the Unix system, it is necessary to predefine the usage of the RS-232 ports. A single port cannot be used as an input port (where a user can login to) and an output port (which a user can not login to, and thus can be reserved for some special purposes, for example, computer-to-computer communication) at the same time. Therefore, in most of the cases, we do not expect that one would reserve many ports for the ITP-CM communication at the cost of less input ports. Thus if several ITP connections are to be established at same time, they must be multiplexed. This is done by the CM layer.

For the following reasons, We chose to take a slightly different implementation approach on our Sun systems, although it would be also possible to use the software described above.

1. Sun's Unix BSD4.2 operating system offers powerful interprocess communication services. It supports some popular communication protocols, such as TCP and IP etc. We chose the TCP as local transport protocol. As it is truly end-to-end, we do not have to have an extra process to implement the additional layer.
2. Unix BSD4.2 can support several TCP connections simultaneously. Thus local virtual circuits are no longer very valuable resources. The CM layer does not have to multiplex ITP connections and is therefore simpler.

We also exploited some special features offered by the system and made the implementation more flexible.

Similar to the situation described above, a centralized transport station is used. However, it has only one standby process. This process mainly manages ITP connection-establishment. When a connection is set up, the standby process forks off another process to control the communication through the newly-built connection. It just waits for another new connection. Because the standby process is only active during the connection build-up stage, it is relative small and needs few resources. The forked-off process dies when the connection is cleared or broken. This dynamic configuration makes it possible to use resources dynamically. As every connection is separately controlled by one process after set-up, it is also useful for error isolation and recovery.

Table 1 shows the result of experiments in which data is transferred between two systems connected by a LocalNet-20, using the ITP-CM protocols. Both sender and receiver use a 9600 bps port. Compared with other similar projects, it seems that the results are encouraging. With the implementation of CSNET at Purdue University [12], for example, an average effective data rate of about 1600 bits per second, was achieved. The results in Table 1 coincide quite well with the equation

$$\text{Time} = 1.07 \times \text{Data (K-byte)} + 0.68 \times \text{Data (K-byte)} + 0.12 \times \text{Number of Packets}$$

The first term is the "pure" transfer time in a 9600 bps link. The overhead introduced by network and protocol

software are mainly composed of two parts. The first part is proportional to the number of data bytes. The second part is mainly caused by the overhead proportional to the number of packets, such overhead is introduced with the heads and tails of data packets, the transfer of control packets, the waiting for an acknowledgement and also the process switching when sending or receiving a packet.

Transferring 40K-byte files between two Sun stations connected to an Ethernet takes, on the average, between several seconds to less than 20 seconds, depending on the packet size used. It is much faster than the example above, where two systems are connected by the LocalNet-20 network. Of course, this is due to the higher transfer speed of the Ethernet. The simpler implementation also reduces some overhead, such as time needed for process switching.

We also interconnected the LocalNet-20 and Ethernet networks. Table 2 shows the experimental results. Because the data transfer rate on the Ethernet is so much faster than on the LocalNet-20, the time for sending a messages from a system on the LocalNet-20 to a system on the Ethernet is not significantly larger than when sending to the gateway, or to another system on the LocalNet-20 network. However, when smaller packet

File Size: 40-Kbyte			
Packet Size (Bytes)	Number of Packets	Time (Sec.)	Data Rate (Bits/Sec.)
64	640	146	2244
100	410	119	2754
128	320	107	3062
150	274	103	3181
200	205	95	3449
250	164	89	3682
256	160	88	3724
300	137	86	3810
350	117	84	3901
400	103	82	3996

Table 1.

sizes are used, the performance will decrease slightly. The reason for this is that more time is wasted waiting for acknowledgements, due to the delay of acknowledgement packets incurred at the gateway.

File Size: 40-Kbyte			
Packet Size (Bytes)	Number of Packets	Time (Sec.)	Data Rate (Bits/Sec.)
64	640	189	1734
100	410	148	2214
128	320	133	2464
150	274	124	2643
200	205	113	2900
250	164	102	3213
256	160	101	3244
300	137	94	3486
350	117	87	3766
400	103	83	3948

Table 2.

Concluding Remarks

This paper introduced the INUZ project to interconnect heterogeneous local area computer networks at the University of Zürich, together with the chosen topology, the communication model and associated protocols.

The design of INUZ was influenced by the features of the local networks we are coping with. They are geographically distributed and heterogeneous in technology and performance. Most of them offer a virtual circuit service. Their end-users include small microcomputers with limited resources.

How successful the INUZ project will eventually be depends on how actively it will be used by end users.

We believe that our approach offers the following advantages:

- It supports heterogeneous networks and systems.
- It supports virtual circuit based networks easily.
- It can cope with most proprietary networks.
- The associated protocols can be readily implemented on small systems.
- It leads to a very simple yet versatile gateway design.

However, the following criticism may be brought forward:

- Yet another protocol: We believe, that we had no other choice, than to design new protocols, because

the existing ones, either do not fit into our environment (mostly connection-oriented networks), nor do they fit into smaller systems.

- We are not using a standard: This may be a problem for users wanting to communicate with external users. However, this could be solved by using a centralized gateway server offering standard long-haul protocol services.
- Not the most efficient: Recall, that these protocols are used mainly for internet communication. The local communication can continue to use whatever protocols may be appropriate. (Our protocols will certainly not be used for a file server application). So the local communication is not affected by our protocols.

The INUZ project has taught us:

- No popular standard protocols exist for what we feel to be, a typical environment: heterogeneous nodes and heterogeneous local area networks. We therefore believe that further research on internetworking is needed.
- It is difficult to implement realtime protocols in a non-real time environment. Not enough operating system and hardware support for implementing communication protocols exist. (Unix BSD4.2 is certainly a step in the right direction, will, however, be only available on a small number of systems).
- Proprietary networks with many protocol layers in ROM of front-ends are cumbersome. These networks are not "open". If one could use the services of the lower layers, simpler protocols could have been developed.
- A formal description and specification of communication protocols should be done before implementation. (We learned the hard way). A special programming language for programming communication software would be most practical.

References

- [1] V. G. Cerf and R. E. Kain, "A Protocol for Packet Network Intercommunication", IEEE Trans. on Commun. Vol. COM-22, May 1974, pp. 637-648.

- [2] C. A. Sunshine, "Interconnection of Computer Networks", Computer Networks, Vol. 1, No. 3, January 1977, pp. 175-195.
- [3] Z. Fan, "Interconnecting Heterogeneous Local Area Networks", First International Conference on Computers and Applications, 1984 (to appear).
- [4] A. A. S. Danthine, "Network Interconnection", Proc. IFIP TC 6 International In-Depth Symposium on Local Computer Networks, April 1982,
- [5] R. Callon, "Internetwork Protocol", Proc. IEEE, Vol. 71, No. 12, December 1983, pp. 1388-1393.
- [6] V. G. Cerf and P. T. Kirstein, "Issues in Packet-Network Interconnection", Proc. IEEE, Vol. 66, No. 11, November 1978, pp. 1386-1408.
- [7] J. Postel, "DOD Standard Internet Protocol", IEN 128, RFC 760, USC/Information Sciences Institute, NTIS document number ADA079730, January 1980, Also in Computer Comm. Review, Vol. 10, No.4, October 1980.
- [8] J. Postel, "DOD Transmission Control Protocol", IEN 129, RFC 761, USC/Information Sciences Institute, NTIS document number ADA082609, January 1980, Also in Computer Comm. Review, Vol. 10, No.4, October 1980.
- [9] A. Bechtolsheim, et.al., "The SUN Workstation Architecture", Computer Science Department, Stanford University, Tech. report, January 1982.
- [10] K. Biba, "LocalNet: A Digital Communications Network for Broadband Coaxial Cable", Proc. COMPCON 81 Spring, IEEE, pp. 59-63.
- [11] M. Stumm. "Interprozesskommunikation auf lokalen Netzwerken: Untersuchungen über Grenzen und Möglichkeiten bei geschlossenen Systemen", Ph.D Thesis, University of Zürich, 1983.
- [12] D. Comer and J. T. Korb, "CSNET Protocol Software: The IP-to-X.25 Interface", SIGCOMM '83 Symposium on Communications, Architectures & Protocols, March 1983, pp. 154-159.

THE WANG PROFESSIONAL IMAGE COMPUTER : AN OVERVIEW

Ahmed El-sherbini, and Mike Smutek
Small systems Development
Wang Laboratories Inc.
Lowell, MA 01851
(617) 459-5000

ABSTRACT

The Professional Image Computer (PIC) designed by Wang laboratories Inc., is a highly integrated office product based on the Wang Professional Computer (PC). It introduces the computer imaging technology to the office automation arena. The PIC can capture, process, and communicate images in real time thus adding another dimension of processing capability to the Wang PC. With the PIC image processing becomes an integral part of the information resources in the office environment.

In this paper we present an overview of the Wang Professional Image Computer. The system's technical features, components, and functions will be described and illustrated.

INTRODUCTION

Traditionally, computer manipulation of images [1,2] was a task that only large and expensive machines could accomplish. That was true for the following reasons : (1) dealing with digitized images means manipulating huge amounts of 'data (more than 3 million bits per one image) which was beyond the capabilities of small machines, (2) the real time performance required in many applications (such as the office) was not possible due to the limited processing speed of the available microcomputers, and (3) the permanent storage for such amounts of data was very expensive.

But with the rapid advancements in semiconductor technology, in high speed microprocessors with large addressable memory, in software engineering, and in image processing techniques and algorithms, and with the decrease of the hardware cost and the enthusiasm of the engineers involved in the project, Wang Laboratories was able to introduce the computer imaging technology in a cost-effective way to the place where it is needed most: the office.

The Professional Image Computer (PIC) is the newest member of the Wang professional computer

product line, and is fully compatible with Wang's VS/IIS, VS/Alliance, OIS, and Alliance systems.

Based on the Wang PC, the PIC includes: a desktop, camera-like, scanner to digitize images from a sheet of paper; a high-resolution monitor capable of displaying the image; and a desktop thermal printer capable of printing the image. A picture of the PIC system is shown in Figure(1). PIC image software permits users to scan, digitize, create, display, alter, merge with text, store, retrieve, and transmit images. Images can also be attached to databases and can be retrieved by keywords. PIC can handle image information that includes pictures, handwritten notes, drawings, as well as text. It also supports all existing Wang PC system software and applications. With PIC, the ability to process images, words, and data, and to communicate and transmit that information locally or remotely within Wang's entire family of compatible office products, is now available in a single workstation. The PIC fulfills the user's needs for image processing capabilities and advanced systems integration [3,4].

THE PIC HARDWARE

Because the design of the Wang PIC is based on the Wang professional computer, the new image hardware permits existing PC installations to take advantage of the PIC imaging capabilities. The base hardware of the PIC consists of the following components :

1) The electronic unit : is the Wang PC electronic unit (standard 5 slot unit or expanded 8-slot unit, which include the 8086 based processor card, the 128 KB of RAM, and the 320 KB floppy disk drive) equipped with:

- * Image monitor interface card
- * 512 KB memory expansion card
- * 10 MB winchester disk drive and control card
- * Scanner / printer interface card.
- * Optional communications cards.

The system cards and connections are illustrated in Figure(2).

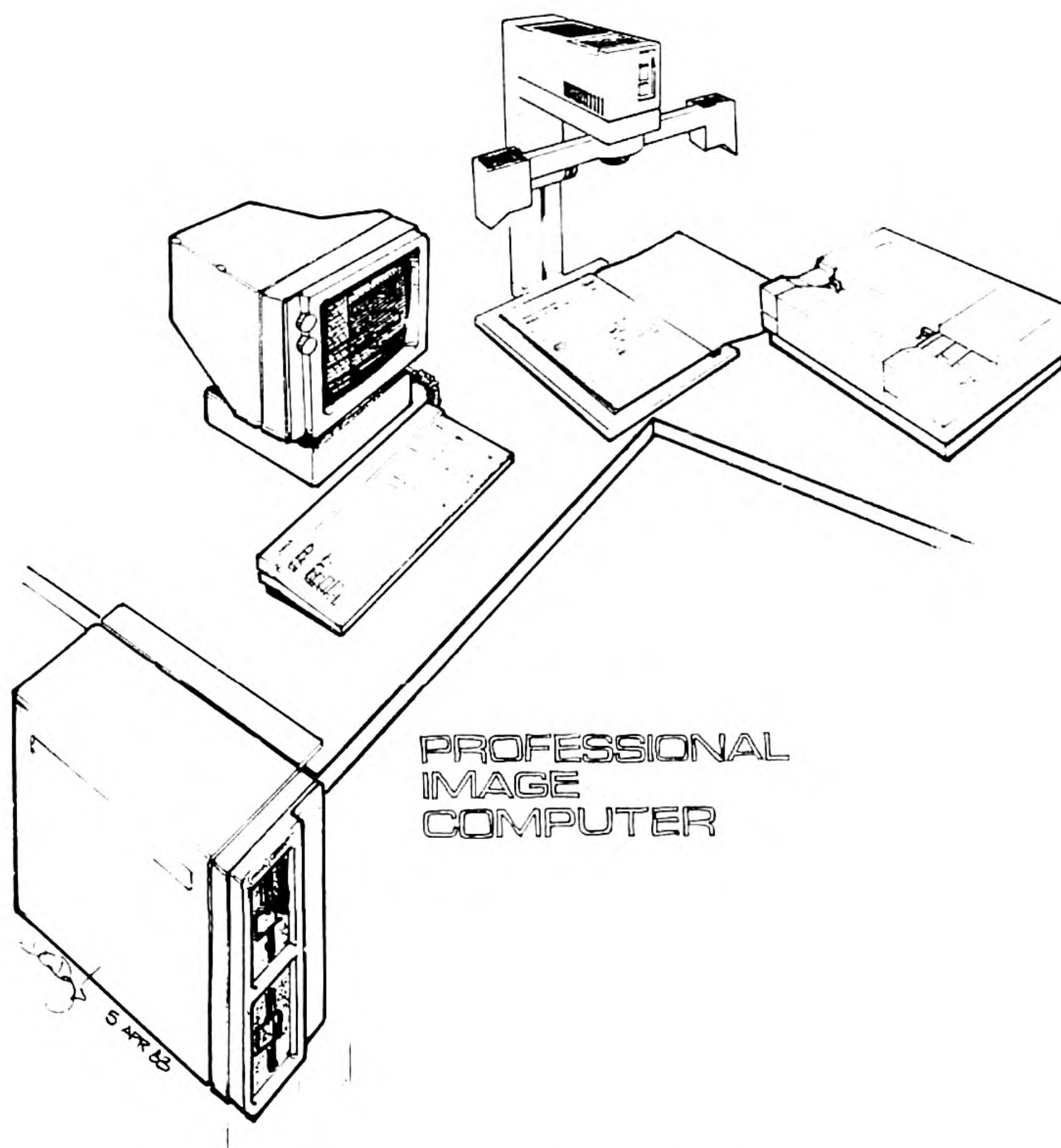


Figure 1: The Wang Professional Image Computer (PIC)

2) Image Scanner : is a CCD, camera-like, desktop scanner that scans a full or a partial page up to 11 by 14 inches in dimensions. It digitizes at a nominal resolution of 200 dots per inch (dpi) in both directions, providing a resolution compatible to CCITT Group 3 facsimile machines [5]. The scanning unit contains two fold-out lamps that provide the illumination necessary to capture the image. The scanner supports two modes of operation: a text mode, and a photo mode. In the text mode the camera is set to scan any document that contains text or typing, and produce a binary (black-and-white) image. In the photo mode the camera will output "half-tone like" images using an "ordered dither" technique.

Dithering is an image processing technique which creates a two level picture that gives the illusion of a gray scale picture by appropriately controlling the spatial density of black and white picture elements [6,7]. This mode is useful for media containing photographs, and it simulates the effect of 63 levels of gray. The scanner is programmed to dynamically adjust its thresholds to compensate variations in document contrast and lighting.

3) High Resolution Monitor : is a 12 inch monitor with a nonglare, white-on-black display. It contains an 800 by 600 dot screen that displays both text and images. Display resolution is 100 dpi. Images are usually displayed as black on a white background, and can be scrolled vertically and horizontally on the screen.

4) Printers : The PIC supports both the Wang thermal printer (shown in figure 1), and the Wang Laser Imaging System model LIS-12. The thermal printer is a compact, desktop unit which creates a high resolution character and image output on draft quality paper at 200 dpi. The laser printer system provides high quality outputs for PIC generated images, text, and documents integrating image and text. It prints at a speed of 12 pages per minute, at a resolution of 300 dpi in both directions. It also supports numerous paper sizes including 8.5 by 11 inches, 8.5 by 14 inches (legal size), and 8.27 by 11.69 inches (for international applications).

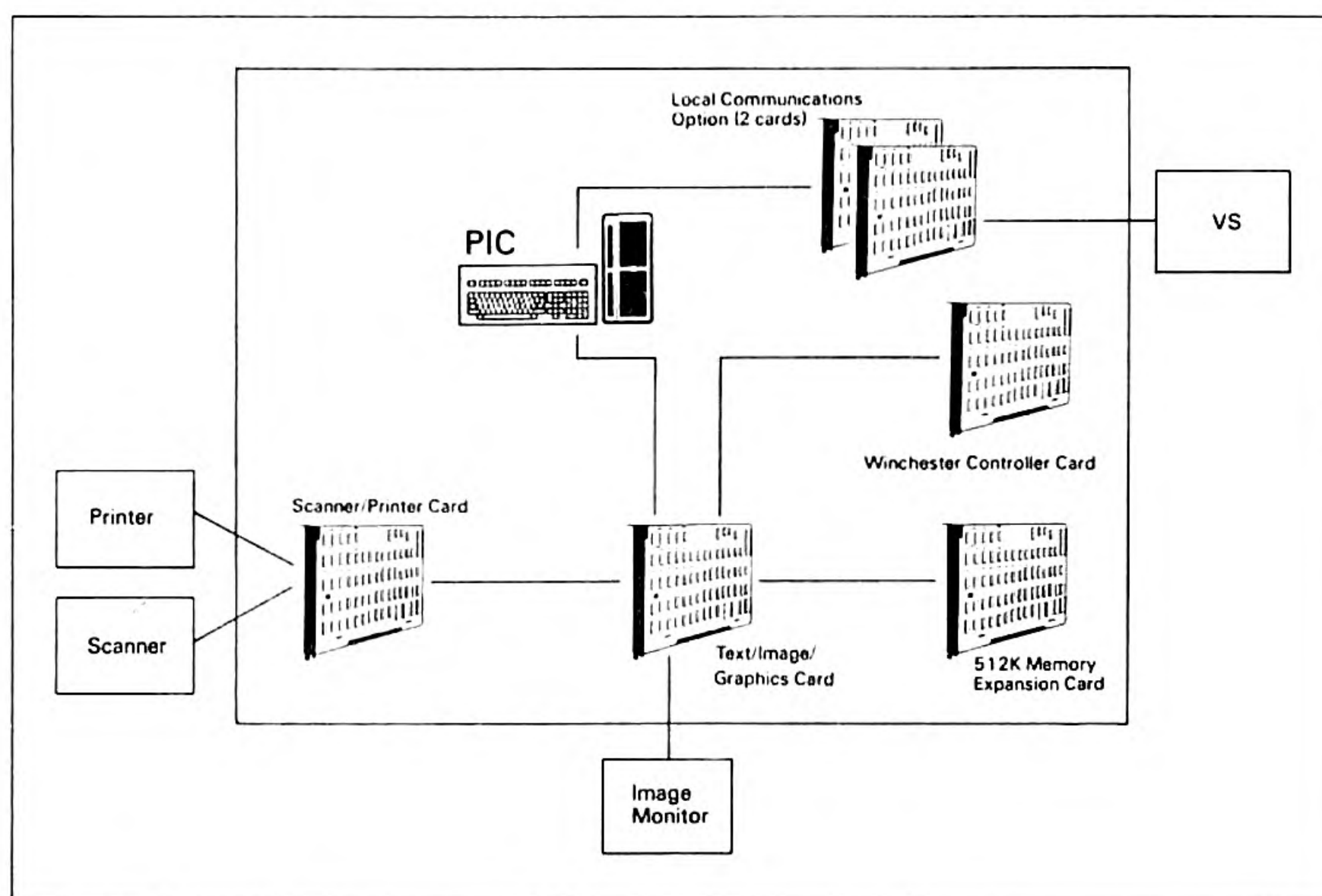


Figure 2: The PIC System Diagram

IMAGING SOFTWARE

The PIC offers several software functions which provide the generic tools required to create custom applications. The available software can be divided into three categories :

- 1) Imaging core software : which includes
 - *Image Processing
 - *Image Composition
- 2) Optional PIC imaging software : which includes
 - *Integrated Word Processing
 - *Integrated Database
 - *Image Forms-Fill
 - *PIC programmer utilities
- 3) PC compatible software : which includes
 - *Wang PC system software
 - *Wang PC Business Graphics
 - *Wang PC Word Processing

The components of the first two categories are briefly described in the following subsections.

Image Processing :

The image processing core software allows the user to scan and organize image information in the form of documents. Each document contains one or more image pages which can be created, displayed, reviewed, printed, deleted, and transmitted or received. Displayed images can be scrolled horizontally and vertically,

darkened or lightened, enlarged or reduced, viewed in portrait or landscape mode (that is rotating the image 90 degrees), and viewed positive or negative. The image scanning, for convenient operation, can be initiated from either the Wang scanner or the keyboard. Each image is compressed before storage or transmission, which reduces storage requirements and allows for faster transmission. The PIC uses the 1-D run-length modified Huffman coding scheme [8], which is the CCITT recommendation T.4 for Group 3 facsimile machines [9]. This makes the Wang PIC compatible with most of the existing facsimile systems. The data flow diagram of the PIC system is illustrated in figure (3).

Image Composition :

The image composition package allows the user to perform composition of image information in a manner similar to "cut-and-paste" operations performed with paper. In image composition, images are accessed only by reference to cut down the storage overhead and leave the original image data intact.

Integrated Word Processing :

This package not only provides the ability to perform standard PC word processing functions, but also permits the integration of PIC-generated text and images. Supported features include full Wang word processing

capabilities, the ability to insert images prepared through image composition into a PIC word processing document, horizontal and vertical positioning of images relative to the text, and the ability to format text either around or directly over image information.

Integrated Notebook :

The integrated notebook is image software added to the existing PC notebook. The Wang PC notebook is a database package which holds simple, unstructured, single record notes. The notes can be queried and listed using word indexing techniques. This new integrated notebook enables the user to enter free-form text description of images along with the associated images into a notebook for later examination. The system can search for the desired image file based on the content of the notebook description.

Integrated Database :

This package provides the capability of integrating images with database information. It is based on the Wang PC database package, which is a traditional database with user-definable record structure, multiple views,

and sophisticated query resolution. Integrated database allows the query of structured data, unstructured text, and associated images. Files created with PC database are fully compatible with integrated database.

Image Forms-Fill :

The forms-fill permits the user to define forms, fields, database, and records that contain image, data, and textual information. The form can be a standard paper form which has been scanned into the system. The package allows the user to control data entry by assigning field types, in free format, anywhere on the form, thus assuring proper information input on the form.

PIC Programmer Utilities :

This package contains image utilities that enable software developers to create specialized PIC applications. It allows user-developed applications, written in compiled Basic, Cobol, Fortran, or Pascal to access system-level solutions to integrate images into external applications. PIC programmer utilities include image access routines for image storage, retrieval, and display.

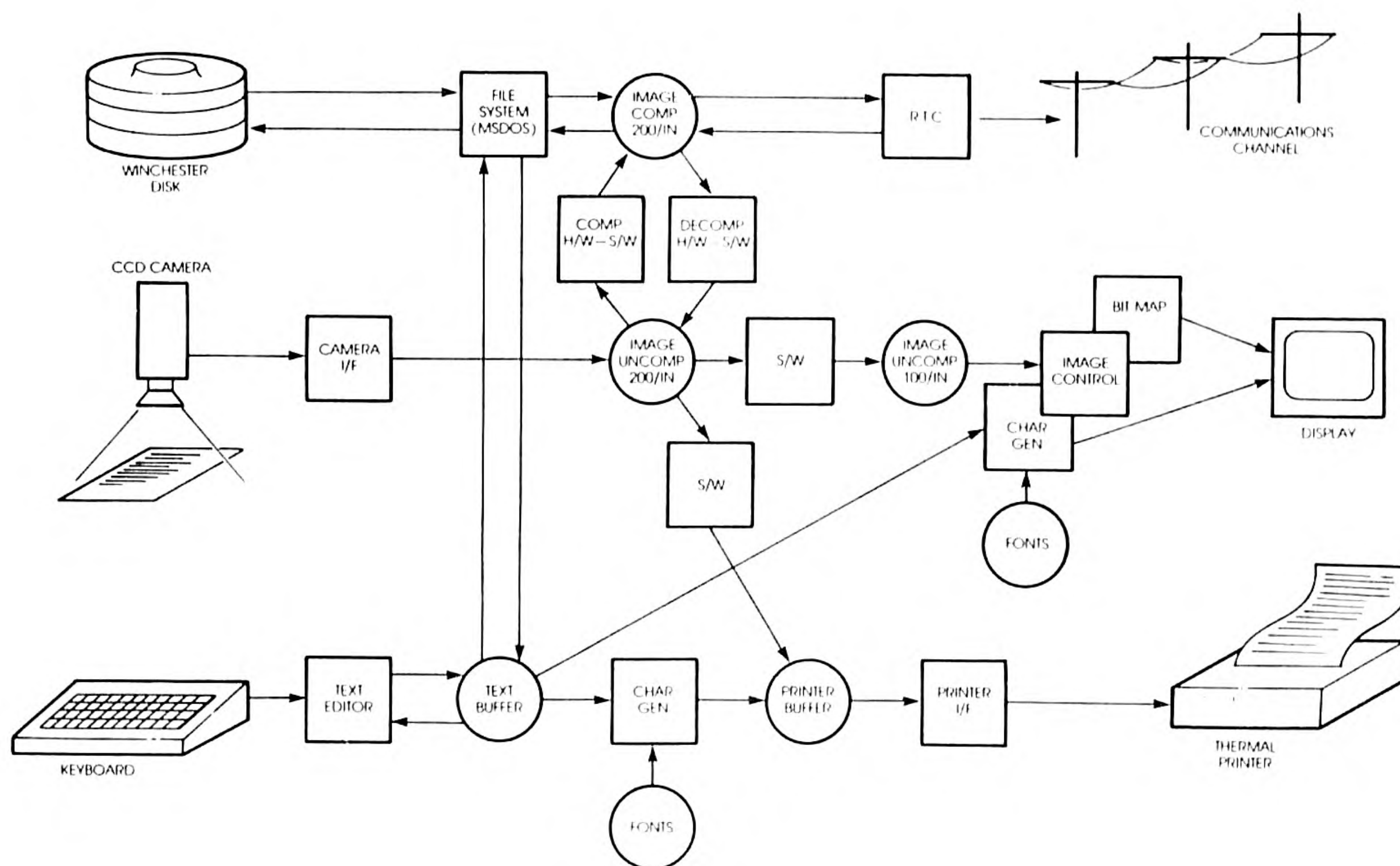


Figure 3: The PIC Data Flow Diagram

COMMUNICATIONS

The advancements in image manipulation offered by the PIC are complemented by new communications options that offers a high degree of systems integration. A properly configured PIC system can scan, transmit, and exchange image information with the entire family of Wang compatible products. Wang provides the following three communications options for PIC image transfer:

Local Communications Option (LCO) :

Any PIC configured with a local communications option can be connected to a Wang VS/IIS, OIS, or Alliance systems. These systems are star configurations with a central master providing file services and satellite terminals. The master and the satellites communicate via shared memory interfaced through a 4 Mhz coaxial cable. This option provides the following functions to the PIC :

- (1) It can use any of the above systems as a file server
- (2) It can emulate a workstation connected to any of the systems
- (3) A PIC without scanner can access shared files which have been created by another PIC's camera.

Figure(4) shows a block diagram of a PIC with local communications option.

Local Interconnect Option (LIO) :

The local interconnect option provides low end networking capability. It allows a number

of Wang PICs to communicate with one another via a local interconnect repeater and share peripherals. In this local loop configuration a series of 5-and 8-slot PICs may share files as well as scanner and thermal or laser printer options across the loop as illustrated in Figure(5). PC systems may also be included as part of the loop to share non-image information. The LIO system uses a modified token passing access method, and a baseband transmission technique. The system's high data rate (2.5 Mbps over an RG-62 coaxial cable) insures the performance necessary for applications involving real time processes. The local interconnect repeater enables up to 24 devices to be configured in one local loop. Also a repeater can attach to another repeater creating a network of a maximum of 255 nodes.

Remote Communications Option :

The Wang Systems Networking Point-To-Point Transport provides remote communications capabilities for PIC-to-PIC transmission of images, data, and text; and includes TeleSend functions and local and remote TeleMail functions. The communications protocol for the transfer of images requires synchronous modems, and a minimum transmission speed of 4800 bps. Switched or leased lines can both be used. Communications occurring over switched lines can support auto-dial using the PC voice attachment and a direct connect modem operating at 4800 bps.

CONCLUSION

Office automation means different things to different people. At Wang Laboratories, office automation is based on the development and application of six information technologies: data processing, word processing, image processing, audio processing, networking, and human factors. The state of the art points to giving the user access to ever-greater processing capabilities through smaller yet more powerful multifunctional systems.

The Wang Professional Image Computer (PIC) is an integration of the six technologies in one system. The PIC is the first product to offer the image processing technology, at the desktop level, to the office in a cost effective way. With PIC, the ability to process images, words, data, and voice and to communicate and transmit that information locally or remotely is now available in a single workstation.

As office imaging computers and workstations become more cost effective, and as more systems and applications are introduced, image processing will become an integral part of the office information processing environment.

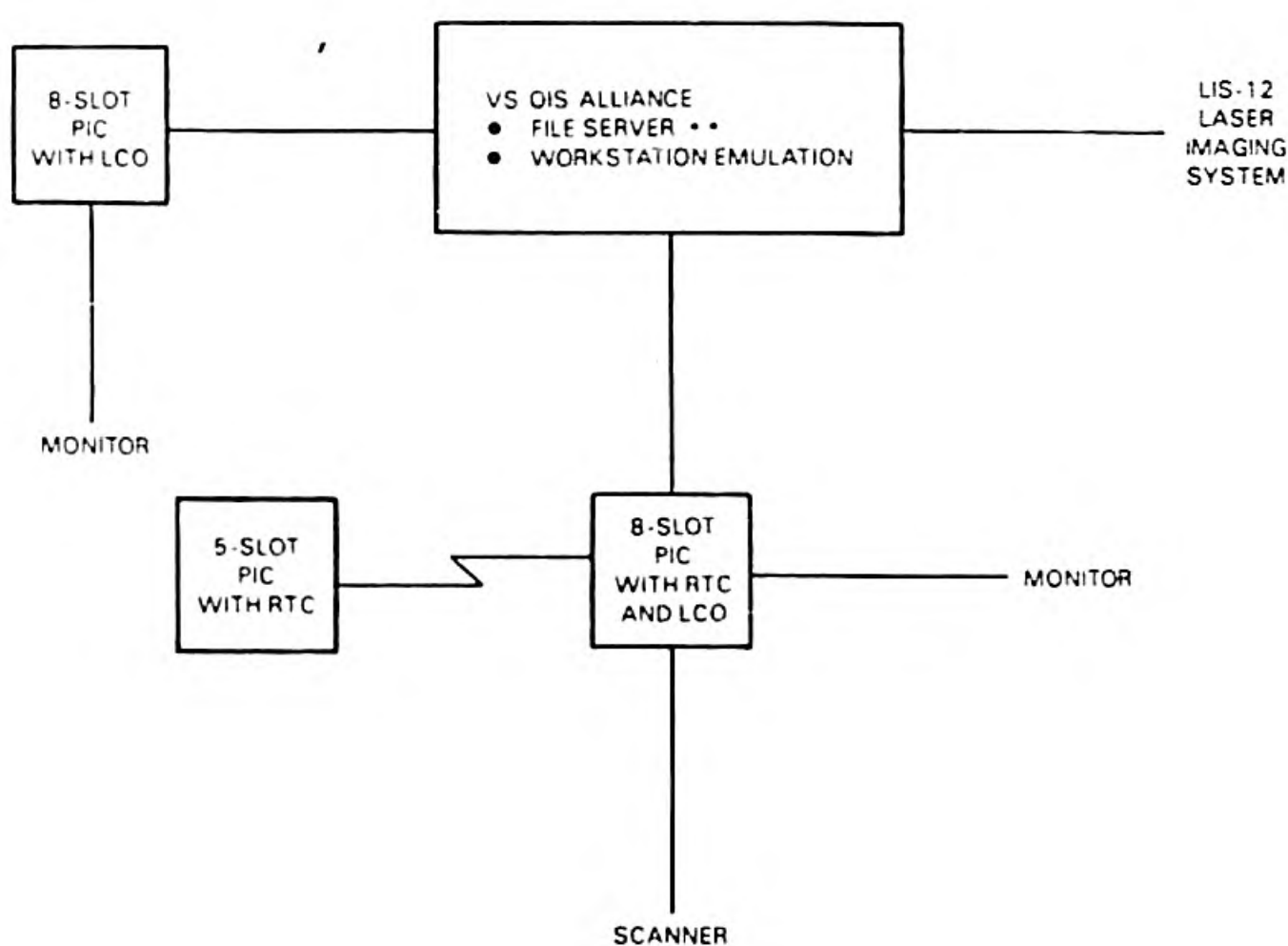


Figure 4: PIC with Local Communications Option

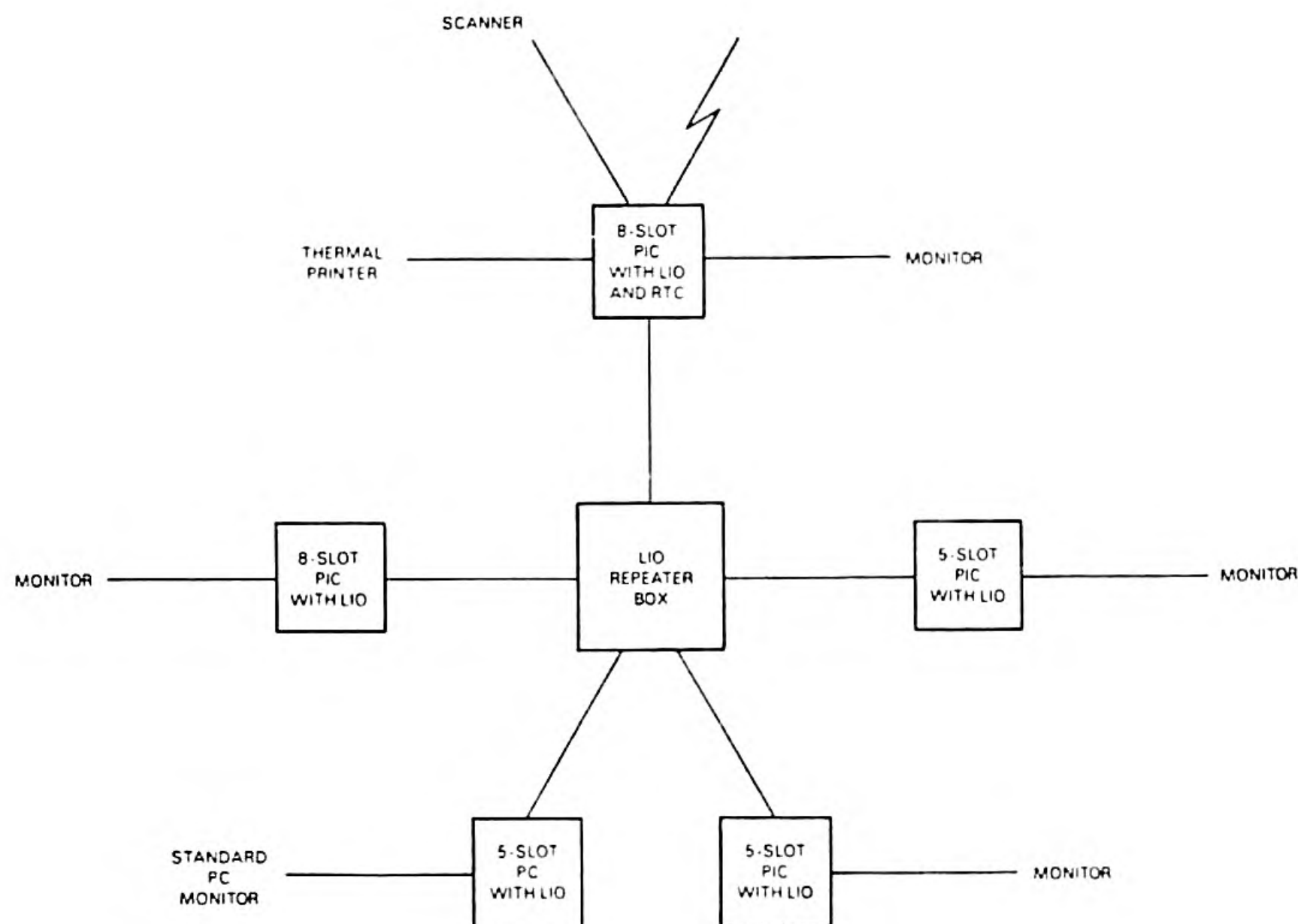


Figure 5: PIC with Local Interconnect Option

REFERENCES

- [1] E. Hall, Computer Image Processing and Recognition, Academic Press, Inc., NY, 1979
- [2] R. C. Gonzalez, and P. Wintz, Digital Image Processing, Addison-Wesley, Reading, MA, 1977
- [3] R.A. Myers, "Trends in Office Automation Technology", IEEE Comm., Magazine, PP. 10-16, Sept., 1982
- [4] N. Naffah, Editor, Office Information Systems, North Holland Publishing, NY, 1982
- [5] J.G. Posa, "A Fax in Every Office", High Technology, PP. 24-29, August, 1983
- [6] D. Anastassiou, and K.S. Pennington, "Digital Halftoning of Images", IBM J. Res. Develop., Vol. 26, PP. 687-697, November 1982
- [7] B.E. Bayer, "An Optimum Method for Two-Level Rendition of Continuous Tone Pictures", Proc. IEEE Int. Conf. on Communications, PP. 11-15, 1973
- [8] D.A. Huffman, "A Method for the Construction of Minimum Redundancy Codes", Proc. IRE, Vol. 40, PP. 1098-1101, Sept. 1972
- [9] R. Hunter, and A.H. Robinson, "International Digital Facsimile Coding Standards", Proc. IEEE, Vol. 68, No. 7, PP. 854-867, July 1980

A COMPARISON OF TWO COMMUNICATION NETWORKS FOR VIDEOTEX SYSTEMS

A. W. Mansky

Jerrold Division
General Instrument Corporation
Hatboro, Pennsylvania

Abstract

A videotex system allows textual and graphical information to be displayed on home television sets under the selective control of the user. An auxiliary electronic device provides the interface between the communication network and the user's television. Trials of two-way videotex systems to date have utilized the telephone network or a two-way cable television system as the communication medium. In this paper, we examine the fundamental attributes of these two communication networks, and the potential of each for becoming the major communication network for videotex.

Introduction

Computing and communication technologies have joined together to produce a new hybrid technology for delivering home-based information and transaction services -- videotex. A videotex system is one constructed for the widespread dissemination of textual and graphical information by electronic means. The information is displayed on normal television sets with an auxiliary electronic device providing the interface to the communication network and decoding the received data into the signals that control the television display.

There are three components of a videotex system. The information providers (IPs) supply the information and services that are utilized by the end-user. There is a wide range of services that might be offered. One IP might supply a service which allows users to retrieve frames (screens of information) that give updated news and weather reports while another IP might provide users with the capability to interact with one another in a multi-user real-time video game.

The second component of a videotex system is the user terminal, which decodes and

displays the received data. The terminal includes a keypad and/or keyboard that permits the sending of menu choices and other information entered by the user to the service provider, database computer, or other users.

Finally, a communications network provides the means by which the information is transferred among the IPs and the users. The two primary communication networks that have been used to date in two-way videotex systems are the public switched telephone network and cable television systems. In this paper we examine the fundamental attributes of these two communication networks and the potential of each for becoming the major communication system for videotex.

The five attributes examined are penetration, transmission speed, interface unit cost, communication costs, and inter-user communication. Each of these impact the nature, scope, or cost of a videotex system. Together, they determine the types of services that may be feasibly offered to the users of the system.

Penetration

Penetration -- the percentage of all households connected to a given network -- influences the economics and the feasibility of a potential service.

Although we take for granted virtually universal telephone service, its initial growth was extremely slow. It took two decades for over 1 percent of U.S. households to get telephone service, and it was nearly 70 years before half of all households had telephones. Similarly, cable television grew gradually at first, taking some 23 years before it reached 10 percent of households.

Historically, the implementation of a new communication technology has taken many years to achieve widespread penetration.

This is especially true for point-to-point technologies using physical communication links because of very high capital and manpower requirements. However, the current projections are that 66 percent of all U.S. homes will be able to be served by a cable TV system (i.e., a feeder cable passes the home) as early as 1985.

Although the vast majority of existing cable systems are one-way, the cable TV industry is moving more toward two-way systems. The FCC has for some time required that all new coaxial cable systems be at least upgradable to two-way operation. It has been estimated that by 1990, cable penetration will be 50 percent of all households, with 30 percent having two-way capabilities. The development of addressable converters, originally developed to prevent basic cable subscribers from receiving free pay TV programs, has provided the means for selective access to videotex services.

It will be difficult and expensive to upgrade many existing one-way systems. However, the high growth rate forecast for new cable TV systems and the opportunity for upgrading existing franchises as they become due for renewal set the stage for the evolution of cable as the second two-way communications link into the home that can be used as the communication medium for an interactive videotex system.

Transmission Speed

The transmission speed of the communications link has a major impact on the quality of the display. High quality text, graphics, and video pictures are acceptable only if the speed of transmission of the data is sufficient to allow the display to appear very quickly on the user's screen. In addition, some services may be partially or completely real-time oriented, as in a multi-user video game, for example. Clearly the speed at which data can be transmitted is a major factor in determining whether or not such services are possible.

To date, non-local computer-to-computer communication has been accomplished via the existing telecommunication facilities. These facilities, especially the switched public telephone network, were usually designed many years ago, with a completely different goal in mind: transmitting the human voice in a more-or-less recognizable form. Their suitability for use in computer-to-computer communication is often marginal at best, but there is rarely an alternative.

Sending data at 9600 bps (bits-per-second) over an ordinary telephone line makes

accurate reception of the original bit stream unlikely. This is due to the bandwidth, power loss, and noise characteristics of voice grade lines. The theoretical maximum data rate for the telephone system is approximately 30,000 bps. While this is an upper bound, in practice, it is difficult to even approach this theoretical limit. A bit rate of 9600 bps on a voice grade line is considered excellent; 300 and 1200 are more typical.

Modems for the public switched telephone network are available at 2400, 4800, and 9600 bps, but the high cost of these units has been a barrier to their use. The high cost arises largely because it must dynamically condition the circuit to compensate for differing line conditions encountered on a typical switched telephone connection. Most often, however, the higher transmission speeds are used on manually conditioned lines (leased circuits). While this reduces the modem cost, it increases the cost of the communications link. Furthermore, because of the limited availability of additional circuits in the telephone local distribution network, these leased circuits could not be used on a wide-scale basis for the local distribution of videotex services.

The cable used in cable television systems, however, has a usable bandwidth that is orders of magnitude more than that of the existing telephone local loops. That, combined with cable's better bit error rate, makes it capable of carrying up to 400 Mbps of data very reliably. A typical cable spectrum is divided into channels; each 6 MHz channel is capable of carrying between 5 and 10 Mbps.

Practical implementations of short-haul cable packet networks have employed speeds in the 1 to 3 Mbps range. Larger metropolitan-area cable communication networks would transmit data over a much wider area and handle a much larger number of users. This can be done through the creation of multiple shared channels, with each channel forming, in effect, its own data communication network. Such a system is being developed that will have a data transmission speed of 128 Kbps. This gives a cable system an enormous advantage over the telephone system in terms of the speed of transmission of videotex data over the network.

Interface Unit Cost

The interface unit provides the physical connection between the user and the communication network. It performs the necessary signal conversion to ensure that

the user's terminal unit is compatible with the network. A decoder is needed to translate the incoming signals into a video display format. The characteristics and the cost of this interface unit is a major factor influencing the potential growth of videotex systems.

Although implementation of digital switching system technology in the Bell System's local office network is expected to start in the mid-1980's, because of the huge investment in the existing telephone network and the impact of divestiture, it is expected that, for at least the next 20 years, the switched telephone network will present an analog interface to the videotex user. Modems, therefore, will continue to be required to interface digital equipment with the telephone network. Adding an estimated cost of \$150 for the decoder portion of an interface unit to a \$500 estimate for the modem, the combined telephone modem/decoder unit cost estimate is \$650.

Interface units for cable-based videotex systems may be thought of as extensions to the existing addressable converters, which currently cost in the range of \$150 to \$200. The added hardware costs for decoding the videotex data and perhaps additional network interface costs contribute to an estimate of \$350 for the total user interface unit.

Clearly, the addition of features, such as a remote QWERTY keyboard, increases the estimated cost of these units. Such features, however, are independent of the type of communication network and were not included in the above estimates.

There is one feature of the user interface unit, however, that is feasible only in a cable-based system. Because of the vastly higher transmission rate using cable (as described in the previous section), it is possible to build a user interface unit that is RAM-based, where virtually all of the software that controls the unit is downloaded each time the unit is used. While this is technically possible in a telephone-based system, an examination of the time needed to do the downloading on the two different systems highlights the problem. Assuming a 300 bps telephone-based system and a 128 Kbps cable-based system, the downloading time for a 64K software package would be 30 minutes via telephone and 4 seconds via cable.

This software downloading capability gives cable an enormous advantage in the degree of flexibility of the user unit. The same hardware can be used for different applications by merely downloading software tailored especially for the given

application. In addition, software updates can be "installed" with virtually no additional manpower, hardware cost, or user inconvenience.

Communication Costs

The cost associated with the connection to and the use of the communication network itself is a significant continuing cost associated with a videotex service. If these communication costs are too high, they may discourage the use of a system that might otherwise be heavily used.

The cost of the communications link in the telephone network is difficult to forecast. Tariff structures have, in the past, often allowed unlimited free local calling but charged for long-distance calls on a time-usage basis. In 1982, long-distance charges for direct-dialed calls during business hours varied from 15 to 38 cents per minute. Thus, the communication cost for a telephone-based videotex system currently depends primarily on the distance between the user and the videotex service user interface computer, and the time of day the service is used.

The AT&T divestiture, however, will contribute to a substantial increase in the costs of local calling. Widespread implementation of a telephone-based videotex system would add further to these costs. Tariffs are structured to take advantage of the unused capacity of the local telephone network by discouraging use of the telephone network during the busiest times. Users of a videotex system would be using their telephones much more often than they would otherwise and their average call length would increase substantially.

Today, the typical toll call cost of accessing a videotex system would be \$100/month (assuming usage of 6 hours/month with 25 percent day, 50 percent evening, and 25 percent night rates), while the cost of accessing a videotex system via a flat-rate call would be zero. The indications are that the breakup of AT&T will result in decreases in long-distance charges and significant increases in local charges.

Typical current monthly charges for cable services are in the \$15 to \$30 range. While upgrading a one-way cable system to two-way would be costly, the majority of the total expenses of the system are shared with the users of the already-developed services. The cable itself and the majority of the cable system network hardware would be used for both videotex and non-videotex services.

The topology of the telephone system is the driving force behind its rate structure. There is the need to minimize peak usage in order to spread the load on the switching offices. A cable communications network is much less load-sensitive. Many different charging schemes may develop for the communication costs for videotex services via cable. Videotex users may be charged per transaction, by time-of-day, per minute connected, by other factors, and combinations of these.

Whatever the communication costs for a cable-based videotex system prove to be, there is every indication that they will be very competitive with the telephone system.

Inter-user Communication

The simple frame retrieval type of service that has been offered by past and present systems does not require any interaction other than that between the user and a main database computer. However, many of the services that could be offered on a videotex system depend on the capability of the communication network to provide for inter-user communication in real-time, perhaps even allowing many users to become involved in a single "conversation".

A real-time, multi-user video game is an example of such a service. Such a game requires the capability to communicate between many users (perhaps as many as thirty or forty) as instantaneously as possible. Both the "n-way conversation" requirement and the desire for a very high transmission rate make the telephone unsuitable for such a service.

Unlike the local telephone loops, however, cable TV does not use a star pattern radiating out from an end office. Instead, everyone in the same neighborhood shares the same cable. High-performance data transmission systems have already been built using the shared cable technology. This topology allows a user's unit to "talk to" all other units or only selected ones, using the addressability feature of the cable system. A group of users may play a multi-user game, for example, with each unit sending out its new "position" every so often. Each unit would also "listen" for the positional data from all of the other units playing and update its screen appropriately. The downloading capability discussed previously would permit each user to obtain the software necessary to participate in the game. The necessary calculations would be performed at each terminal; in this way, inter-unit communication could be reduced to the

sending of positional data alone as just described.

Surveys of users of existing videotex services consistently show that game-playing and inter-user "chatting" are among the services most used and enjoyed. Unlike the telephone system, a cable television system has the topological and transmission rate characteristics that are needed to support advanced inter-user communication services.

Conclusion

The public switched telephone network clearly has the advantage over cable in degree of penetration. Although cable is growing rapidly, it may never reach the virtually universal penetration that the telephone enjoys.

In the other four attributes examined in this paper, however, cable holds the advantage. The most dramatic differences are in the transmission rate and inter-user communication characteristics. Along with cost, these have a profound impact on whether the typical household will be able to afford to use a videotex system, what types of services will be offered, and indeed whether or not a two-way videotex service is able to be feasibly offered at all.

In summary, a comparison of the attributes of the public telephone network and a cable television system as the communication network for videotex reveals that although the telephone has nearly universal penetration, a cable television system compares very favorably in its cost and offers unique capabilities that will permit the offering of new and exciting videotex services. These services are likely to move videotex from its current small-scale limited-area trials into being a widespread home communication and entertainment service.

References

- [1] Ball, A., et.al. "Videotex Networks", IEEE Transactions, December, 1980.
- [2] Chitnis, A. and Costs, J. "Videotex Services", IEEE Transactions on Consumer Electronics, March, 1979.
- [3] Martin, J. Viewdata and the Information Society, Prentice-Hall, 1982.
- [4] Sigel, E., et.al. Videotext: The Coming Revolution in Home/Office Information Retrieval, Knowledge Industry Publications, 1980.

[5] Tanenbaum, A. Computer Networks,
Prentice-Hall, 1981.

[6] Tydeman, J., et.al. Teletext and
Videotex in the United States,
McGraw-Hill, 1982.

A DATA-STATIONARY PIPELINED MACHINE

Ikram E. Abdou

Electrical Engineering Department
University of Delaware

ABSTRACT

Two basically different philosophies are encountered in the control of pipelined machines; one is the time-stationary control, the other is data-stationary control. In this paper, we will discuss the data-stationary control concept with emphasis on its application in image processing systems. We begin with a survey of the basic characteristics of an image processing architecture, and define a system that fits these features. We introduce the data stationary control concept and discuss its advantages and limitations. Finally, we develop some of the hardware needed for the image processing applications.

INTRODUCTION

Many image processing applications involve extensive computations that have to be completed in a relatively short time. A survey of functions needed in such applications is given in reference 1. The hardware needed for these functions has some basic characteristics. In the following sections, we discuss these features, and introduce a hardware that is optimized for such applications.

We begin with a summary of the basic features of an image processing architecture. Most of these concepts are discussed in detail in references 2 and 3, interested readers should refer to these two excellent surveys.

1. Pipeline vs. Parallel Architecture

In a pipeline machine, a function is split into small tasks each one of them performed in a separate piece of the hardware. These pieces are usually different. In parallel processing identical hardware apply the same functions on different pieces of data. Image processing applications require a combination of the two concepts. An efficient machine has to be pipelined. If higher performance is needed, an array of such machine has to be assembled thus producing a parallel array of pipelined machines.

2. Multifunctional vs. Unifunctional Pipeline

A multifunctional pipeline system is designed to execute several sequences of operations. While a unifunctional pipeline executes single sequence of operations⁴. Both approaches exist in image processing. A machine based on the unifunctional concept usually consists of different pipelines each dedicated to a specific function. Such an architecture is fast but expensive. In this paper we will develop a multifunctional pipeline processor.

3. Static vs. Dynamic Pipeline

If the reconfiguration of a multifunctional pipeline can take place any time during processing, the machine is considered to be a dynamically configured pipeline. If reconfiguration of the pipeline happens only while the pipeline is not processing any data, the machine is called a statically configured pipeline⁴. Since most of the image processing applications involve vector processing, it is better to use a statically configured machine to avoid the complexity of the dynamically configured approach.

4. Microprogramming vs. Hardware Controlled Architecture

In a microprogram controlled machine, the major logic devices are controlled by a sequence of instructions that directs the hardware to perform the required tasks. In a hardware controlled machine, random hardware logic is used to direct the machine to perform the various tasks. The flexibility of the microprogram controlled machine makes it more suitable for the multifunctional pipeline systems.

5. Special Purpose vs. General Purpose Approach

Although most of the functions needed for image processing are repetitions of some basic arithmetic operations, there are some functions that require branching and interrupt handling. Also there are

functions that are scalar in nature. The hardware needed to handle these functions is basically different from the hardware used for iterative vector processing. Designing a machine that accommodates these opposite requirements is a difficult task and the resulting system is usually a bad compromise. A better approach is to build a machine that consists of two basic sections, one of them handles all the interrupts and branching functions, in addition to all scalar processing. While the other section is a special purpose pipeline that is optimized for vector manipulation.

In summary, the architecture we propose for image processing applications is a parallel array of pipelined machines. The pipelines are multifunctional, statically configured devices, with a microprogrammed control hardware. The architecture is optimized for vector processing. A block diagram of such a system is shown in Figure 1. The central processor is a sequential machine that handles all the interfacing to the host processor. If the command received requires vector processing, it will be passed to the pipeline section of the machine. The pipeline section consists of a buffer memory which receives data from the main memory through an input/output interface. A processing controller produces a sequence of instructions, that is implemented by the processing unit.

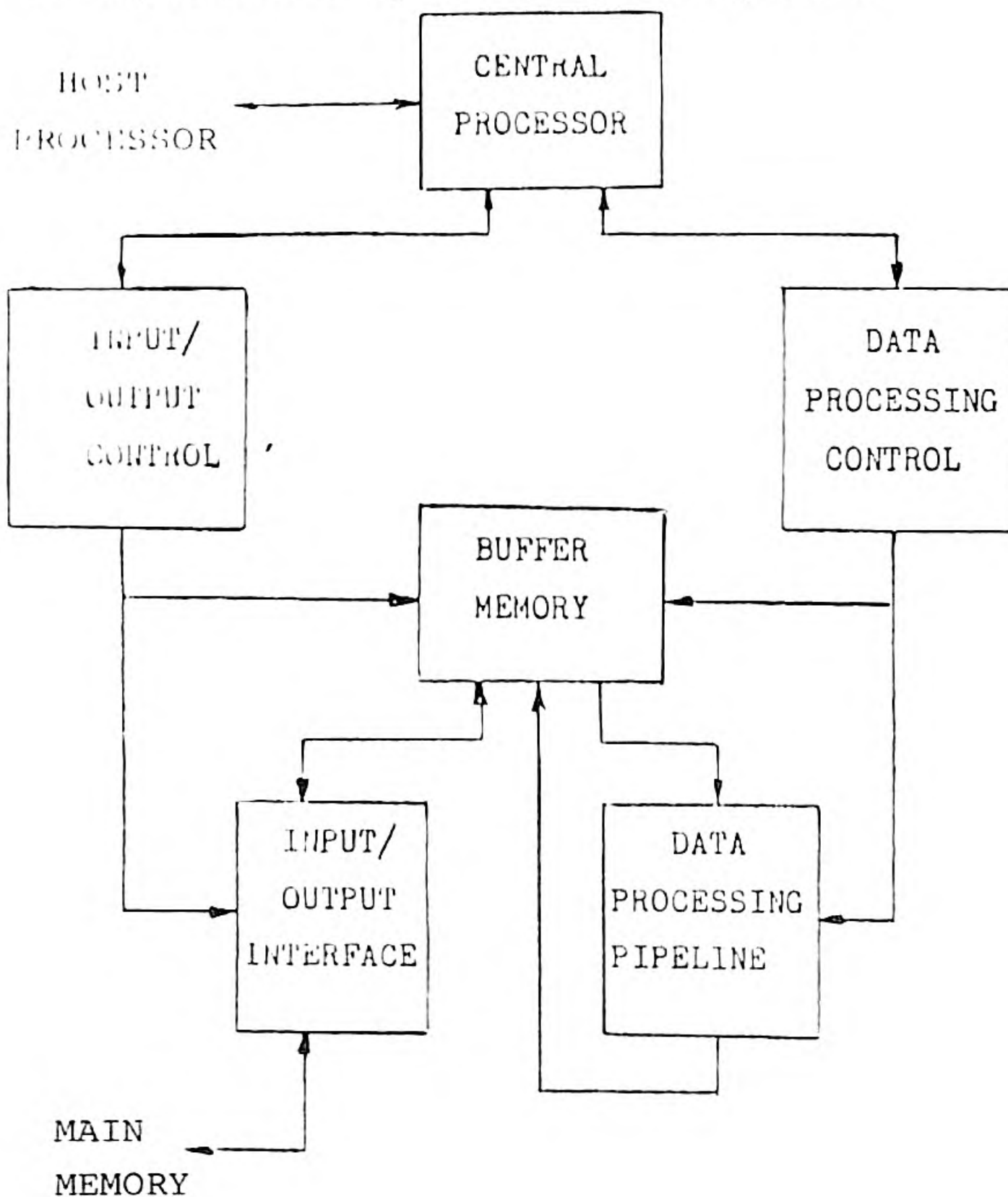


Figure 1. Block Diagram of Proposed System.

After this general discussion of the system architecture, we will focus our attention on the methods of implementing the control signals in the processing unit. There are two basically different philosophies that can be used to control a pipelined machine; one is the time-stationary control, the other is the data-stationary control. The following section describes these two concepts.

TIME-STATIONARY AND DATA STATIONARY CONTROLLED PIPELINE

The time-stationary control is the traditional method of hardware implementation, in which the control signals for the entire pipeline are derived from a single source. In this mode, each instruction is executed in a single cycle of the machine. In the data-stationary machine, every instruction follows the data through the pipeline providing the control signals at each stage as needed². Examples of both methods are shown in Figure 2. The hardware consists of data address generators, data memories, a multiplier, and an arithmetic/logic unit. The connections between the various components are the same for both the time- and data-stationary systems. The basic difference is in the way the control signals are distributed to the hardware. In the time-stationary mode, the output fields of the control register go directly to the pipeline control inputs. In the data-stationary mode, the output fields of the control register pass through a sequence of delay registers before reaching the pipeline control inputs. Every field of the instruction is executed at a different cycle, thus matching the execution time of the data.

The advantage of the time-stationary system is that it requires less hardware. Its disadvantage, which becomes more obvious as the level of pipelining increases, is the complexity of the programs needed to optimize the use of such a machine. On the other hand, a system based on the data-stationary control concept produces more natural and easy to write microprograms. Also the code written for such a machine is concise because the same instruction is used to load the pipeline, to repeat a loop, and to empty the pipeline². This feature is more useful in image processing applications, since most of its functions involve repetitive code. Another advantage of the data stationary system is that it minimizes long distance communication for the control signals⁵. This is important in VLSI design. To benefit from all these advantages, we need to define a system with a data path that closely fits a wide range of applications². This is possible in image processing because the functions needed in this field are of the

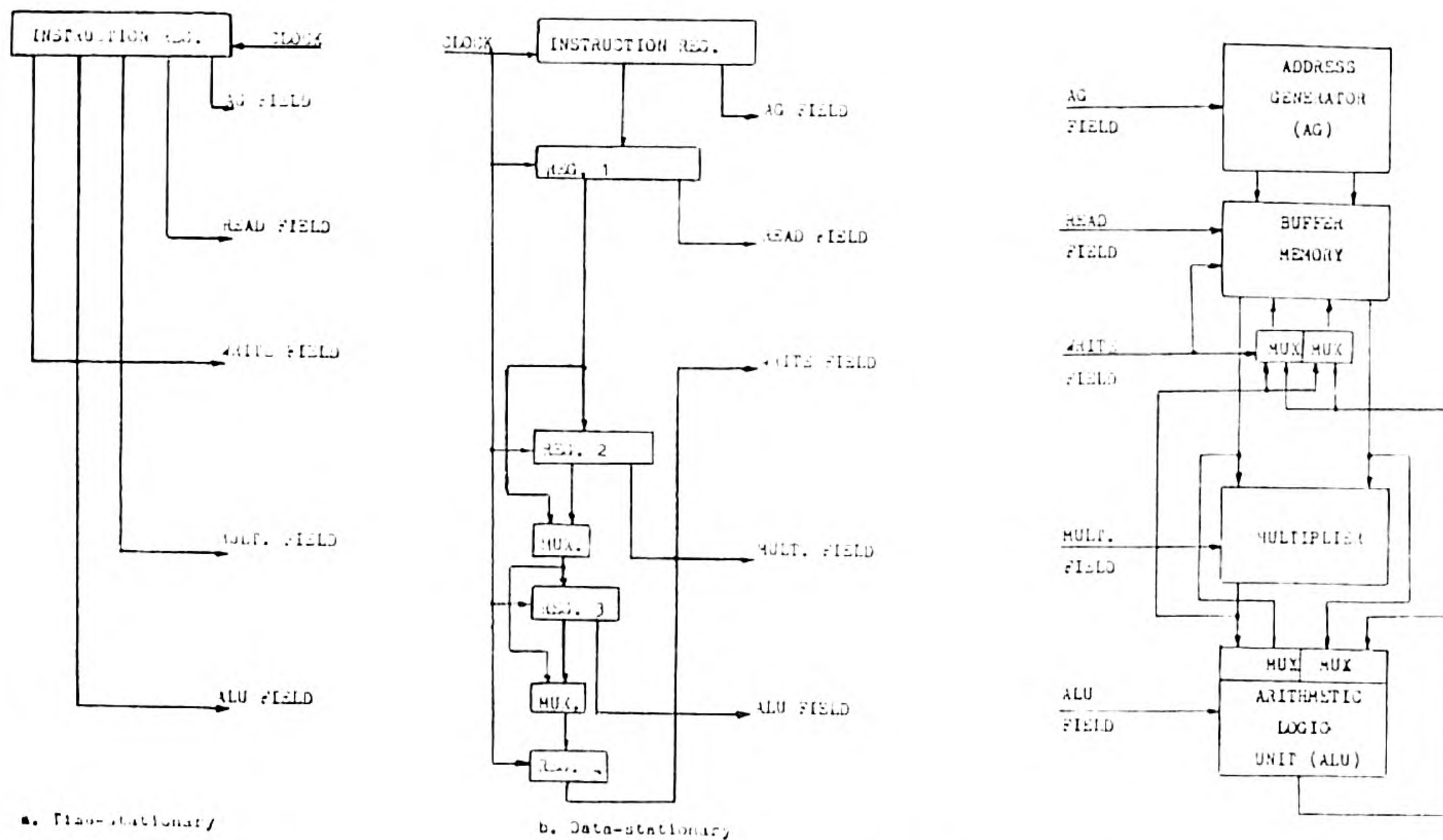


Figure 2. Time-Stationary and Data-Stationary Control Models

vector processing type. The hardware of a data-stationary system should be a superset of all the hardware that may be needed. An optimum path for any given application can be defined using multiplexor selections in both the data and instruction sections. Example of such a hardware is given in reference 1, where the various functions needed in image processing were used to define an optimum data-stationary architecture.

Finally, to complete the discussion of the data-stationary pipeline we will define the main features of an address generator module that may be used in the image processing applications.

ADDRESS GENERATOR MODULE

It is important to note that while images are by nature two- or multi-dimensional, most of the hardware storage elements are one-dimensional. This mapping problem is critical for any successful address generator design. As an example,

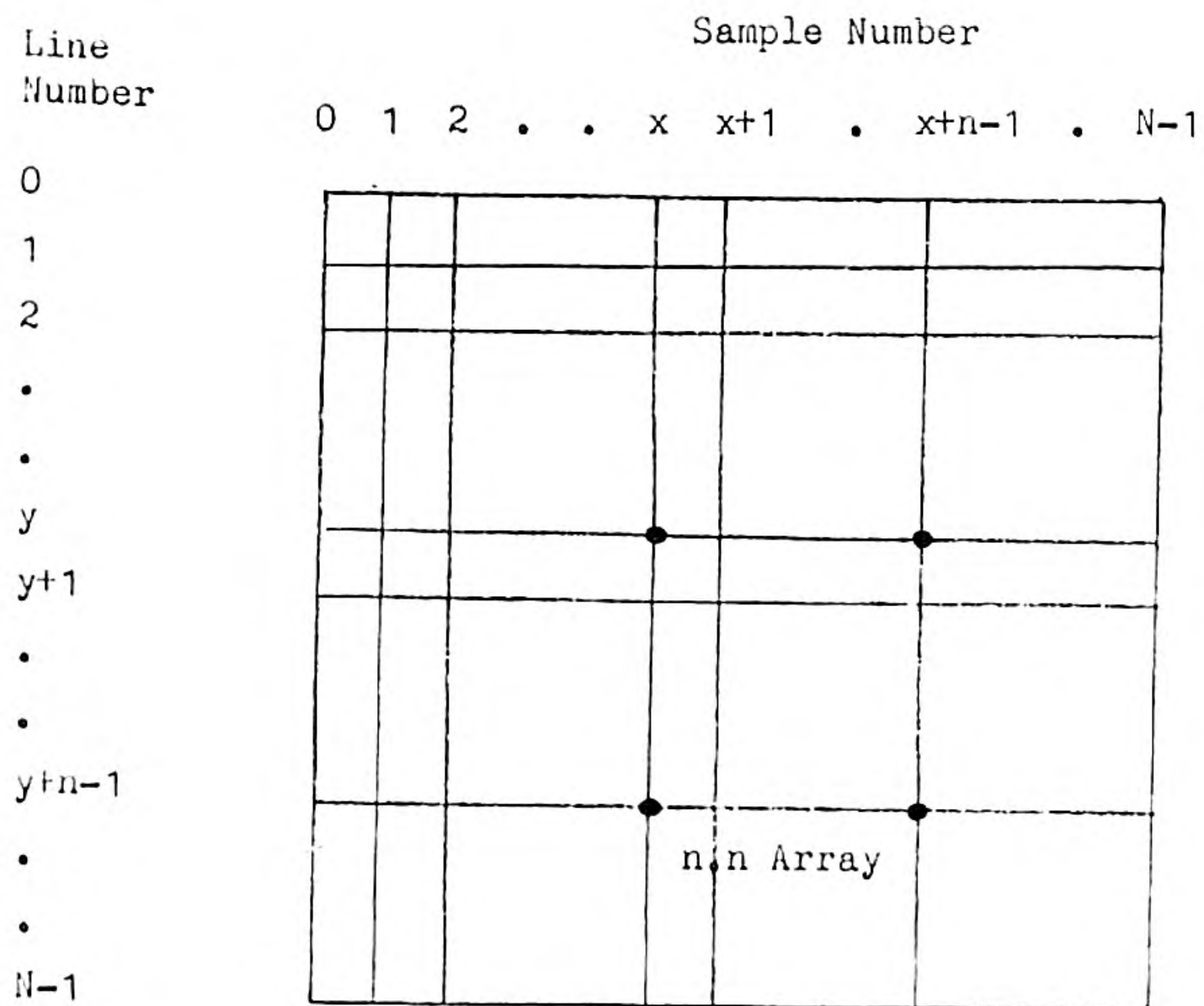
to perform an $n \times n$ convolution on the image shown in Figure 3.a, we store few lines of the image in the pipeline buffer as shown in Figure 3.b. The image addressing scheme for this function has to produce the sequence

$$0, 1, 2, \dots, n-1, n, n+1, n+2, \dots, n^2 - 1 + n, \dots$$

To produce such a sequence, we need the hardware shown in Figure 4.a. In addition, we need to generate addresses for the convolution coefficients and for writing the results. Since all three addresses have to be generated simultaneously, we propose a system of three address generators, two for reading data and one for writing data. The connection for these address generators is shown in Figure 4.b.

CONCLUSION

In this paper we have defined the basic characteristics of the hardware needed for the image processing applications.

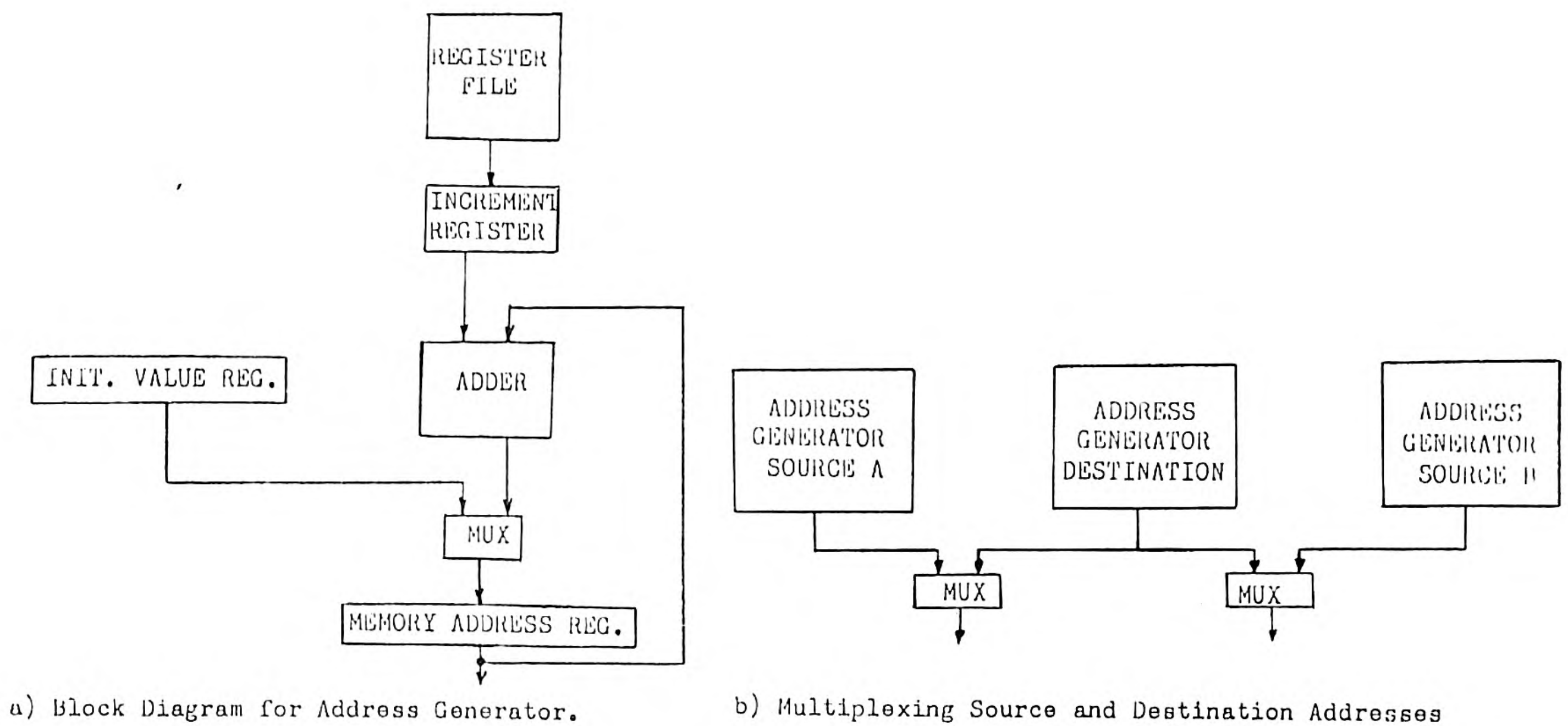


a) N.N Image

Image Pixel	Memory Address
$f(y,0)$	0
$f(y+1,0)$	1
$f(y+2,0)$	2
.	.
.	.
$f(y+n-1,0)$	$n-1$
$f(y,1)$	n
$f(y+1,1)$	$n+1$
.	.
$f(y+n-1,1)$	$2.n-1$
$f(y,2)$	$2.n$
.	.
$f(y+n-1,n-1)$	$n.n-1$
.	.
$f(y+n-1,N-1)$	$N.n-1$

b) Memory Mapping

Figure 3. Mapping from a Two-Dimensional Image into a Memory



a) Block Diagram for Address Generator.

b) Multiplexing Source and Destination Addresses

Figure 4. Address Generators Scheme

The system with the optimum performance is a parallel array of pipelined machines. Each pipeline is a multifunctional, statically configured, data-stationary device. The pipelines do not accommodate branching instructions or interrupts, and the design focus is on vector processing only. We believe that making these decisions is critical to the optimization of the hardware. The system we propose can be used in other applications such as signal processing and arithmetic number crunching.

REFERENCES

- [1] I.E. Abdou, "A Pipeline Machine for image Processing Applications," Proc. Intl. Conf. on Parallel Processing, (1983), pp 255-257.
- [2] P.M. Kogge, "The Architecture of Pipelined Computers," McGraw-Hill, (1981), pp 334.
- [3] C.V. Ramamoorthy and H.F. Li., "Pipeline Architecture," Computing Survey," Vol. 9, No. 1. March 1977, pp 61-102.
- [4] S.P. Kartashev and S.I. Kartashev, "Adaptable Pipeline System with Dynamic Architecture," Proc. Intl. Conf. on Parallel Processing, (1979), pp 22-230.
- [5] J.R. Burger, "Data Stationary Instructions as a Way to Minimize Long Distance Communications in VLSI," Proc. Intl. Conf. on Parallel Processing, (1983), pp 548-553.

DATA STRUCTURES ON DATA FLOW COMPUTERS: IMPLEMENTATIONS AND PROBLEMS

Susan Samet
Maya Gokhale

University of Delaware

This paper examines hybrid mechanisms which have been developed to solve the problem of implementing data structures on dataflow machines. These methods require that the data delivery system transmit a pointer to the structure rather than all the elements of the array. The array, then, is stored in a separate memory, and only the pointer travels on the arc. Schemes used by the Utah DDM, the MIT data flow machine, and the Manchester University machine are presented. Implications for the use of complex data structures on data flow machines are also discussed.

INTRODUCTION

Data Flow is a simple, powerful model of computation in which a program is represented as a data dependency graph in order to make explicit the maximum amount of parallelism in the problem representation. The nodes indicate operations to be performed and the arcs indicate transmission of data from a producer node to a consumer node. There is no central global memory.

A data flow computer interprets such a graph. The computer must have some mechanism for delivering data values to operators. Although a straightforward implementation of this function is adequate when the operands are of primitive data type (character, real, integer), transmission of structured data (such as an entire array) to an operator is impractical, particularly when only one element of the array is required.

Over the past few years several groups have investigated computer architectures for dataflow that are based on parallelism and asynchrony implicit in the dataflow model. In this paper we look at the MIT Data Flow Computer, The Manchester Data Flow Computer, and the Utah Data-Driven Machine. The data structuring mechanism chosen for each machine and their implementations are examined. Also, the problems and limitations involved in these implementations are discussed.

There are two basic ways for the data structures to be implemented. In the first method, every single component of a particular structure flows through the graph on the same data path. In other words, there is a token for every value within the structure. In the second method, the structure is stored in a memory and is represented by a pointer. Thus, a single token travels on the data path and serves as a pointer to the memory in

which the structure is stored. The implications and advantages/ disadvantages of these methods are discussed in relation to the dataflow models presented.

The only data structuring mechanism which has actually been implemented is for the DDM1, where it is part of the architecture of the machine. The Manchester group has developed a language called MAD (MANchester Dataflow) which supports data structures. Many of the requirements of the Manchester Machine to support data structures were established during the language implementation effort, and some of these features may be included in future prototypes.

THE M.I.T. DATA FLOW MACHINE

Overview

The M.I.T. Data Flow Machine exhibits a packet communication organization. The machine is divided into five major units connected by channels through which these packets are sent [see Appendix]. The machine language, called Dennis Dataflow Nets (DDF), is a directed graph representation in which both data tokens and control tokens can enable an instruction. An instruction is enabled, or fired, when all of the input tokens it expects have arrived at the node indicating the operation to be performed. The arcs of the graph can only be occupied by one token at any given instant. Another feature of the M.I.T. processor is that operands are sent to instructions rather than having instructions fetched from memory. A detailed description of the machine structure can be found in [8],[13].

The M.I.T. Structure Controller

Data flow structures on the M.I.T. Data Flow Machine (MIT DFM) are acyclic binary trees whose nodes are either primitive values or pointers to other nodes. The depth and length of a structure (where length is defined as the number of branches of the top level of the tree) must be finite. Every node of the tree has a selector indicating the node's position in relation to the root of the tree. The selector is a bit string which tells whether to follow a node's left or right pointer when traversing the tree from its root to the node to be selected [2].

Only two machine-level structure operations have been defined for the MIT DFM. The first is the select operation which takes a particular structure and selector, and returns the object (value) at the specified selector:

```
select(structure,selector)
-->[object at point in
    structure indicated by
    selector given]
```

The second is the append operation, which takes a structure, selector, and a value to be added to the given structure at the specified selector, and returns the structure containing the added object:

```
append(structure,selector,object)
-->[structure with given object
    added at selector position]
```

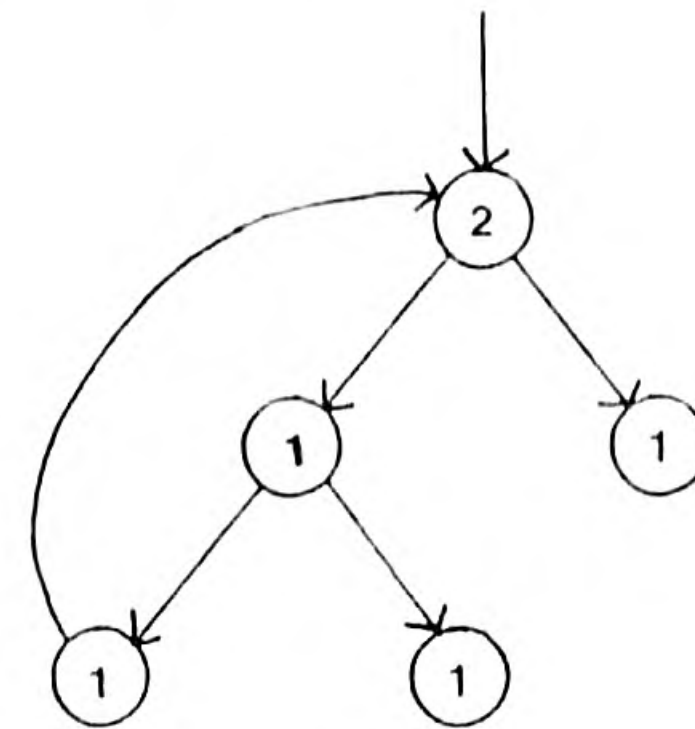
Since dataflow follows the principle of single assignment (a variable can have only one value), any append operation must result in a new structure being created. This makes the structure controller's job of memory management rather complex. Clearly, it is inefficient to copy an entire structure whenever a value is altered. Therefore, the structure controller is designed so that substructures can be shared between structure values without violating dataflow semantics. This is done by keeping a reference count for each node of a structure which indicates the number of pointers that reference that node in processors or in other nodes. If a node's reference count is zero then the node is not being accessed and can be returned to free storage. In addition, the nodes that were pointed to by the deallocated node need to have their reference counts decremented.

A special case arises when a node's reference count is equal to one. Only one pointer to the node exists and it is possible to change the structure in place without affecting any other structure values. Thus, when an append operation is being performed and the reference count is one, no copying is necessary. Finally, if a node's reference count is greater than one, meaning that the structure is being shared, copying of the structure cannot be avoided if the structure is to have something appended. However, it is only necessary to keep one physical copy of the top level of the structure in structure memory [2].

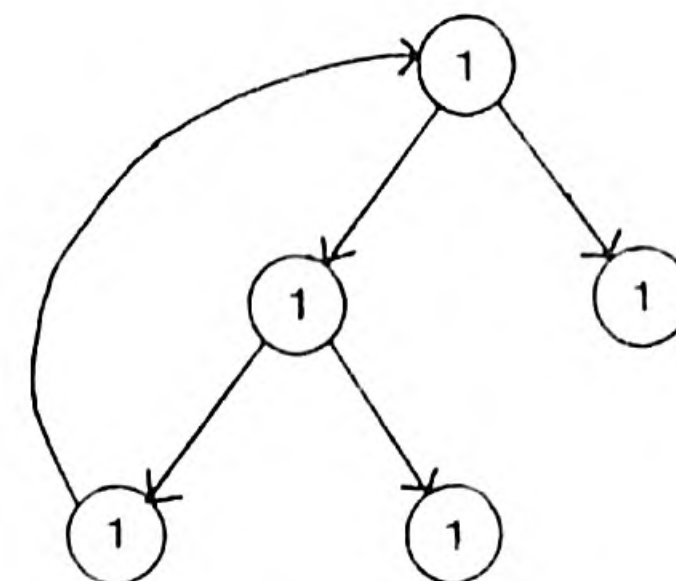
It is this reference count method that prevents the use of recursive data structures (cyclic trees). This is illustrated by the case where the pointers to a cyclic structure have been deleted. The nodes of the cyclic structure still have reference counts of one and would never be returned to free storage. The graph in figure 1 depicts this situation.

Even with acyclic structures it is difficult for the structure controller to free child nodes when the reference count of the parent node becomes zero. Whenever this occurs, it is possible that the reference counts of two other nodes

may become zero, and management becomes a recursive process that is difficult for the structure controller to handle since it has no stack memory. The structure controller deals with this problem by reducing the right node (B in figure 2) when the parent node is to be returned to free storage. The left pointer of the returned node remains and the right pointer points to the next free cell. Thus, the process becomes iterative, since reclaiming one cell can only lead to reducing one more node's count to zero [2].



(pointer to tree exists)



(pointer has been deleted)

Figure 1 - Deleting pointers to a cyclic structure

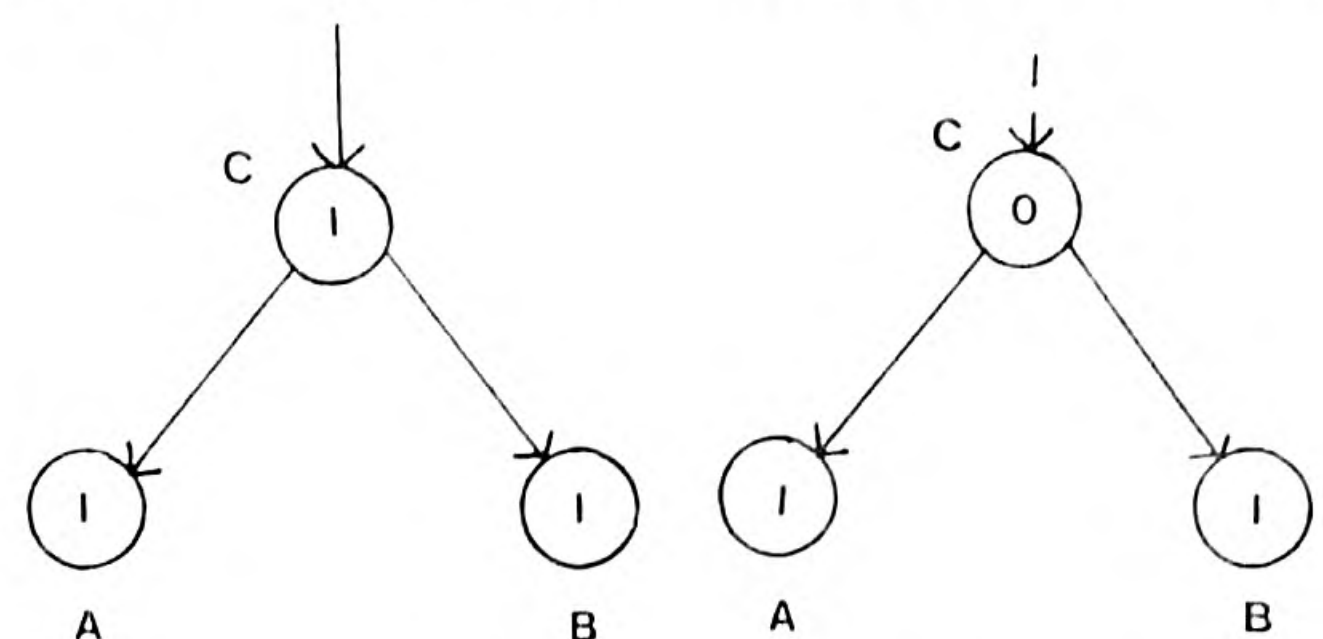


Figure 2 - As a result of C's count being reduced, A and B should be returned to free storage.

The structure controller is able to minimize copying via the reference count method, as explained previously. To guarantee that the structure controller will perform operations on data structures correctly, Ackerman [3] states that it must be proved that no cycles will be created and that the append operation can be defined using the car, cdr, and cons (pure) LISP operations, which produce no side effects. The reader is referred to [2] for the append and select algorithms, and for a detailed discussion of the design of the controller and its relationship to other modules of the M.I.T. Data Flow Machine.

Proposed Extensions

A hypothetical data flow computer which is very similar to the actual M.I.T. DFM has been designed by Rumbaugh, and many of the proposed extensions can be applied to the improvement of data structure handling on the M.I.T. DFM.

One idea is to provide each processor with its own structure memory, to avoid conflicts in requests for access to a structure, and also to speed up structure operations. This is also motivated by the fact that components of structures are often accessed in sequence. It is also possible to take advantage of locality of reference by having structure pages which would be swapped in and out of the structure cache.

If this were to be implemented, reference counts could be stored in a cache memory as well. Rumbaugh points out that altering reference counts could be overlapped with fetching Structure Memory words for a particular structure operation.

Another improvement in data structure accessing would be to take advantage of what Rumbaugh terms common-structures, which are structures that can be changed concurrently, but which cannot be copied. This can be done if the concurrent updating is done on elements with different indices. A forall construct which adheres to this rule has also been proposed.

Rumbaugh has presented many other ideas for the design of a parallel architecture for data flow machines, yet many fundamental problems remain. Even before the specific details of the implementation of data structures can be worked out, some more basic issues need to be resolved, such as the actual design of a structure cache and a specific paging scheme for bringing structures from the storage into the structure caches.

As concluded by Rumbaugh, "The most serious gap in knowledge needed to build an actual Data Flow Machine is the absence of concrete program characteristics -- the amount and kind of concurrency within algorithms, the intensity of memory usage, the size of structures and the kinds of structure access used..."

THE UTAH DATA-DRIVEN MACHINE

Overview

The Utah Data-Driven Machine (Utah DDM1) differs vastly from the M.I.T. Data Flow Machine because the program and machine organization of the former are based on recursion, which is forbidden in the latter.

The DDM1 is a tree-structured multiprocessor. Each computing element is a processor-storage unit pair which can be the root of a subtree of identical elements, as well as part of a larger tree [7],[13]. The machine language, referred to as a DDN (Data-Driven Net) representation, is a directed graph where data tokens are list structures. Communication between instructions is accomplished solely via data tokens since, in this representation, there is no distinction made between control and data. Thus, there are no control tokens. The arcs of the directed graph are FIFO queues. A detailed discussion of the architecture and machine language can be found in [7].

The Storage Model (TSM)

As mentioned above, data tokens are list structures which can represent complex data structures such as plexes or trees, or regular data structures such as vectors and arrays. There are functions which concatenate, decompose, and index these list structures.

The Storage Model itself is a list structure, thus reflecting the overall recursive hierarchy of DDM1. TSM is a variable field size storage representation which allows for dynamic memory management. The idea behind TSM is to "provide a location independent method for dealing with an arbitrary structure of variable length fields" [7]. For example, since DDN processes represent separate computational tasks which will increase and decrease in size during execution, they can exist as Storage Model dynamic files.

In TSM a field is defined to be some number of characters delimited by left and right parentheses. A field can contain elements that are also fields. A TSM structure, then, is a field which may be comprised of other well-nested fields.

Particular data structures (e.g. scalars, tuples, strings, lists, n-ary arrays) are stored as generalized trees. The program representation of a DDN, like the Storage Model, is itself a hierarchically nested structure. The directed graphs which represent the DDN are folded into this generalized tree structure shown in figure 3.

Similarly, the entire data flow program, including any of its subprograms and all of its instructions, is represented as a well-nested character string which can easily be mapped onto a

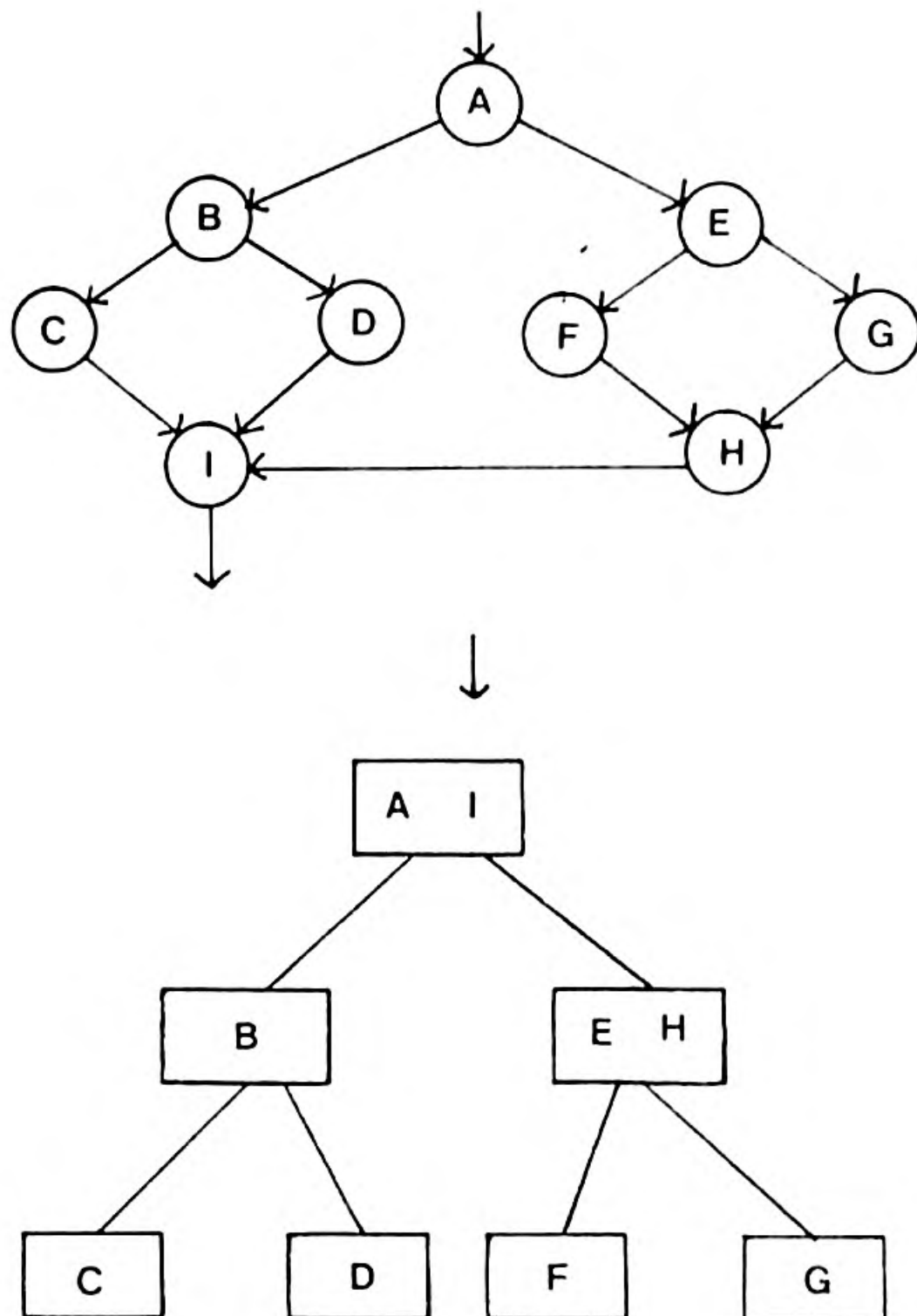


Figure 3 - Generalized Tree Structure of a DDN
generalized tree.

Other data structures are defined using fields (see figure 4). For example, a file is a field containing subfields, where the first subfield is the descriptor for the rest of the subfields. A field which has no subfields is a record. The fields which constitute these data structures can be ordered or unordered. Unordered fields have a name and are accessed by that name, whereas ordered fields without names are accessed through indexing. Fields may also consist of substructures with no depth constraints. In this case, the field is indexed by an access vector whose elements are either index numbers or names, depending upon whether the field is ordered or unordered. Finally, Utah DDM1's atomic storage unit (ASU) (see Appendix) is a TSM file system which is capable of performing file operations such as initialization of memory contents, and insertion and deletion of fields [7].

The DDN's are actually made up of a data item file and a net description file. One approach is to make one of the files dynamic and the other static. This choice provides more insight into some problems that must be handled when implementing data structures. Since dynamic file elements are destroyed when a cell fires, some copying is unavoidable. If large data structures are being used then the data item file should be static so copying can be minimized. The data item file then becomes a static resource that concurrent

((ordered vector) (element value 1) (value 2)
(value n))

(a) representation of a TSM vector

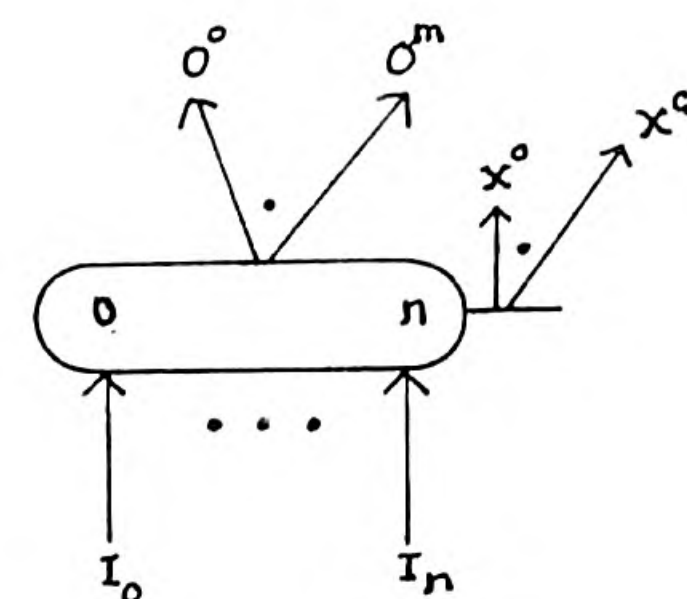
(((ordered matrix)
((ordered row 1) (value 1)...
(value n))
((ordered row 2) (value 1)...
(value n))
.
.
.
((ordered row m) (value 1)...
(value n))))

(b) representation of a TSM m x n matrix

Figure 4 - TSM Data Structures

processes can share. However, this leads to the possibility of access conflicts that arise when structures are shared [7].

The Utah DDM1 has an arbiter cell which guarantees first-come-first-served access to a structure. This use of this cell is essential when shared resource DDN's are being used. Figure 5 shows the arbiter cell in detail. (Subscripts indicate data paths which may receive different valued tokens; Superscripts indicate data paths which will carry identical valued copies of output data items.)



I_j : inputs
 O^i : outputs
 x^K : index outputs
Firing Set: at least one input
 I_j
Cell Function:
 $O^i := \text{first } I_j; x^a := j$
for all i, a .
(In case of a tie any input
 I_j present is chosen.)

Figure 5 - DDM1 Arbiter Cell

In order to see how the DDM1 supports recursive DDN's and DDP's it is necessary to look at some of the other cell types used in the data driven nets. The SYNCH cell synchronizes parallel streams of data. For example, if a process needs input that has been produced by parallel processes then these inputs would be collected at the SYNCH cell. When all had arrived they would be passed into the process. Also, it is possible to detect whether or not the process is ready to execute by first determining if the SYNCH cell itself is firable. Without this capability the steps taken to decide when a process is ready to execute become complex since the notion of global information is not applicable to the DDN representation.

Next, the OPERATOR cell performs arithmetic operations, relational operations, etc. Operator cells are divided into classes which are distinguished by the number of inputs and outputs that are required to fire the cell. The specific operator definition is created by placing the operator symbol (e.g. '+', '*') inside the cell. Finally, the CALL cell invokes a DDP. One important characteristic of the CALL cell is its resemblance to the OPERATOR cell, since the CALL cell performs a user-defined function while the OPERATOR cell performs a built-in function. This similarity permits the two cell types to be interchanged in a DDN program [7]. These three cell types are depicted in figure 6.

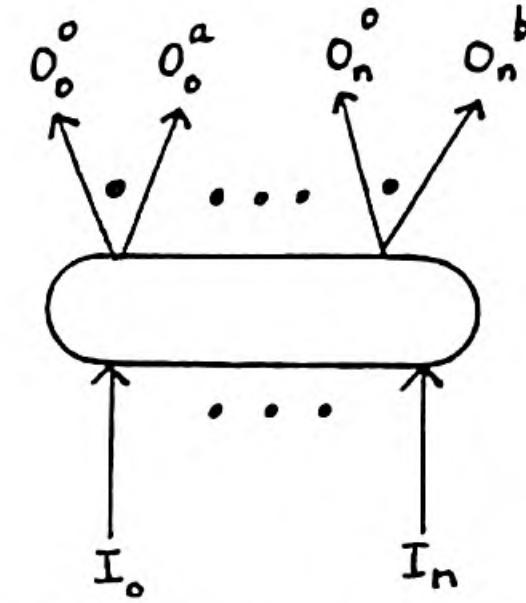
At this point the form of a DDP can be described. The DDN is delimited by input and output SYNCH cells which synchronize the input and output paths and provide local state information. From figure 7 it can be observed that DDP's behave like the OPERATOR and the CALL cells. In particular, a DDP of any DDN that executes a function F may replace an OPERATOR cell that performs F [7]. This property is quite important because it allows recursively defined DDN's and DDP's. In effect, a calling mechanism is defined by this substitution.

THE MANCHESTER DATA FLOW MACHINE

Overview

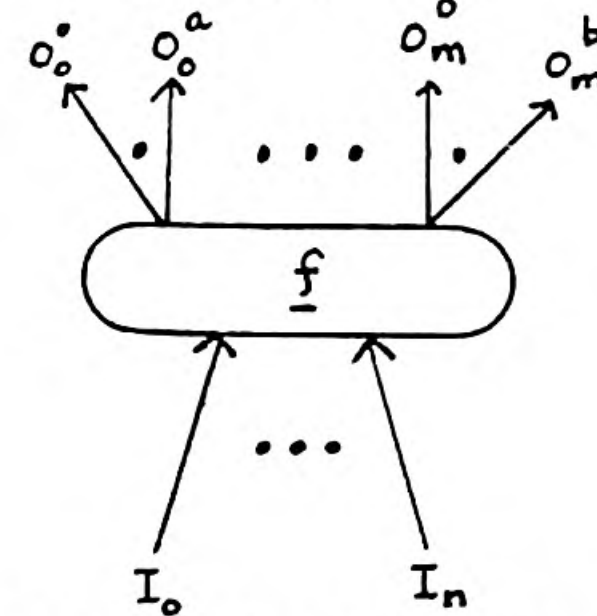
The Manchester Machine displays a packet communication organization with token labeling and token matching. The machine is comprised of five main units which are illustrated in the Appendix. For the purposes of this paper the Matching Store Unit will be of primary concern due to its role in the handling of data structures. In the machine language directed graph representation, each node has at most two input arcs and two output arcs. The reason for this will become clear when the matching unit is explained.

A token contains three fields, the first being a data type field to ensure that operators will have input tokens of a compatible type. The second field is a label field which may have an index (ix), an activation name (an), and an



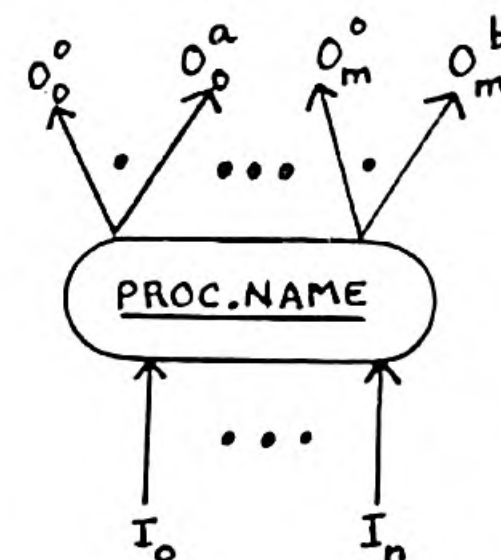
Cell Function: for every i,
 $j := O_j^i := I_i$

(a) SYNCH cell



Cell Function: for every i,j:
 $O_j^i := f(I_0, \dots, I_n)$

(b) OPERATOR cell



Cell Function: for every a,b:
 $O_b^a := \text{PROC.NAME}(I_0, \dots, I_n)$

(c) CALL cell

Figure 6 - Three DDN Cell Types

iteration level (il). The index is used to access elements of a data structure. The activation name is similar but differentiates function or procedure calls. The iteration level separates instances of passes through loops. The third token field is the destination field which contains a node address, an input point, and a matching function needed when the token enters the matching unit. The destination field allows functions or procedures to return results to the proper nodes [13]. Figure 8 illustrates a token's components.

An important characteristic of the token

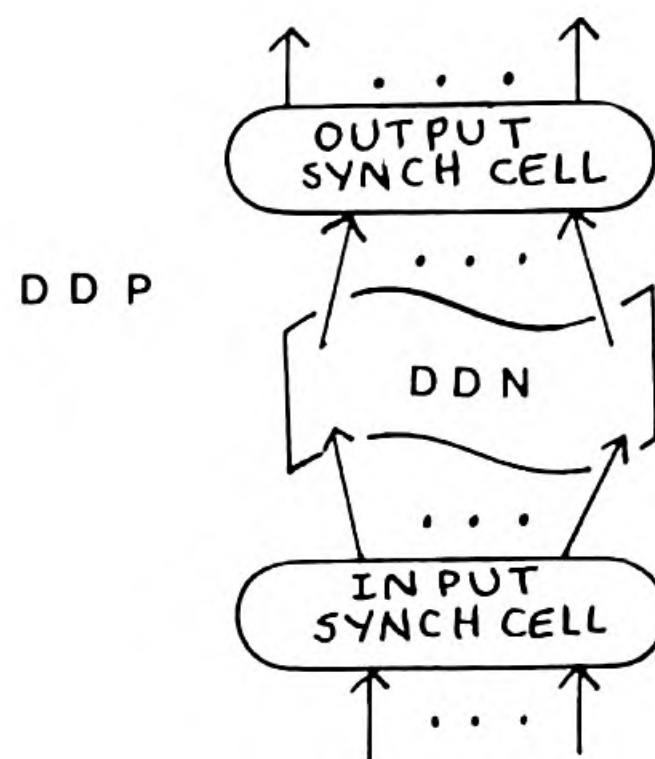


Figure 7 - Data-driven Process

```

<normal token> ::=
                  <datatype><label>
                  <destination>

<label> ::=
          <index><color>

<color> ::=
          <iteration level>
          <activation name>

```

Figure 8 - A View of a Token

label is that it can be changed or created. There are nodes used to set or change these fields, or to generate them. Often, the iteration level and activation name are jointly referred to as a node's color. The color of a node is what permits a graph to be re-used, thus allowing recursion and providing a solution to the reentrancy problem.

Matching functions specified in the destination field of a token indicate what action will take place when the token reaches the input point of its destination node. If the node only needs one input token to fire then a Bypass function would be used. Otherwise, an Extract/Wait (EW) function is used to match the present token with a token with the same label that would reach the node's second input point. When two tokens are matched they enter the node and cause it to fire.

There is another matching function, called Preserve/Defer (PD), which is similar to the EW function but after a match occurs the token waiting for a match can be kept in store and can be accessed again. The matching is done associatively, and the matching function determines subsequent actions [6].

In the next section, the use of these functions and of the token labeling scheme and their relationship to the implementation of data structures will be explained in greater detail. The information in that section is based on [6]. Many of the data structures discussed have been

implemented on the Manchester Machine in the single-assignment language called MAD (Manchester Dataflow). Many of the details of specific implementations at the machine level and of the syntax of the language will be omitted, although examples of MAD constructs will be provided.

The Handling of Data Structures on the Manchester Machine

Extensive examination of conventional data structures and of their suitability for implementation on the Manchester Machine has been done. This was motivated by the viewpoint that Pascal-like data structures have been extremely successful in representing concepts from the real world. Several of these implementations will be presented, along with problems they present. Extensions made to enhance the feasibility of these implementations when using a single-assignment language will also be discussed. Recursive data structuring will also be described.

Arrays and Records

The token labeling scheme allows one-dimensional arrays to be represented as a fixed number of tokens traveling on one arc with index fields signifying a token's position in the array. Yet, in keeping with the single-assignment rule, each element of the array can have its value defined only once. This makes the selective definition of elements of an array (in random order) difficult because it is not possible for a compiler to detect violation of the single-assignment rule. Checking at run-time would not be efficient, and checking to make sure that two statements do not change the same array element (by looking at the indices of all pairs of assignments) is not possible since the concept of memory is lost.

As a solution, a "for..use.." construct was developed. This construct defines the value of an entire array:

```
A := for I use e(I)
```

In the assignment, 'e' is an expression whose value depends on I and defines the value of the I'th element of array A. The range of I is defined by A's index type. The expression 'e' can be evaluated for each I-value in parallel. This construct is useful when all array values are being defined at the same time. An extension of this exists, so that a single element, or several elements, of an array can be updated:

```
A := as B with I := e
```

Here, arrays A and B are the same except that A's i'th element has the value of e.

Records are implemented by designating an arc of the directed graph for each record field, which

is simply a named value. Variant records could be represented the same way as long as the number of arcs is greater than or equal to the maximum number of record components.

Streams

The stream is an extremely suitable structure for data flow computing and the goal of achieving a high degree of parallelism. A stream is a structure whose elements have a total linear ordering but do not all necessarily coexist. Concurrency can be exploited because the processes of consuming and producing streams can be overlapped in a parallel, pipelined manner. This is because the operations performed on streams consume or produce (or both) the streams in the linear ordering defined on them. As with arrays, tokens with a unique index representing each stream element travel on a common arc. (This is referred to as a token stream and is not to be confused with the data structure itself.) One important difference, however, is that the size of the stream need not be known. The end-of-stream condition (eos) must be detected in this case, and this can be done by having a special end-of-stream token.

There are stream operations which are just like operations performed on lists. For example, `first(S)` returns the first element in a stream `S`. Similarly, `rest(S)` returns all elements but the first. The specific implementations of these stream operations can be found in [5].

When using streams with certain data flow language constructs several problems arise. For example, consider the loop construct, within which an operation is being performed on a simple value and a stream. Token labeling requires that a loop constant be replicated for each loop pass so it can be identified with its corresponding iteration level. However, stream elements are distinguished by an index and not an iteration level, as would be a sequence of values being produced during loop execution. In MAD there exists a `REPLICATE` macro node which makes copies of a constant for each stream element, ensuring that the elements are used in order of their indices. Also, each time a stream is passed around the loop, each element must be tested for satisfaction of the loop terminating condition and sent accordingly.

As a result of such complications, several constructs involving streams have been devised. The "return all" construct produces a stream element on each iteration pass without the need to pass the stream around the loop. For example, the following loop returns a new stream `[f(1),f(2),...,f(n)]`:

```
str := declare i, str1:integer;
      init i := 1;
      while i <= n do
        new i := i + 1;
        str1 := f(i)
      return all str1;
```

In MAD, a convert sequence-to-stream node (STX) replaces a token's index by the iteration level in order to implement this construct. A related construct is the for each construct, which consumes stream elements in index order. A convert-stream-to-sequence node exchanges a token's iteration level and index.

Next, the "but" construct allows stream elements to be produced with the added flexibility that elements do not have to be produced on each iteration pass. A special value, denoted by `lambda`, indicates that no element is being added to the stream being produced. For example, altering the previous example so that `f(i)` is created when some boolean function `b(i)` is true yields the following:

```
str := declare i, str1:integer;
      init i := 1;
      while i <= n do
        new i := i + 1;
        str1 := if b(i)
                  then f(i)
                  else lambda
      return all str1 but lambda;
```

A `FILTER` macro node takes a stream and values to be removed and returns the resulting stream.

Finally, the "next" construct extends the "for each" construct so that stream elements do not need to be consumed on each pass. A boolean stream is tested to determine which iterations will require a stream value, and "next s" yields an element of stream `s` if the corresponding stream element is true.

Up to now, only one method of incorporating data structures into the Manchester Data Flow Model has been discussed. That is, the values of components of data structures have been represented as individual tokens. However, this pure data flow method is not always efficient. Recall, for example, that when an array is represented as a token stream, accessing a single component involves many operations. This situation is magnified when the array is used in a loop or in a function.

For this reason, an alternative method is considered, in which structures are held in storage and represented by a token which points to the structure. We have already observed how the Utah DDM and the M.I.T. Data Flow Machine have implemented this method.

The Matching Store

The Matching Store handles the job of memory management, and holds the structures as well. The two basic storage operations are STORE and SELECT. The STORE primitive takes a token stream, stores it, and returns a pointer to the stream. This pointer is then used by the SELECT primitive to access individual stream elements. The implementation of these primitives would require an extension to the current architectural design of the Manchester Machine, but Bowen has proposed a way to simulate conventional storage in the matching store by using the Preserve/Defer matching function mentioned earlier. STORE and SELECT take the form of macro nodes.

There are several steps in implementing SELECT. First, an array is held at the input point of a set-color-and-destination node (SCD), and together they make up a storage node. Next, an accessing token with the same index as the element to be selected and with the PD matching function is sent to the control input point of the storage node. A copy of the selected element is then sent to its appropriate destination, so that the stored array is not changed in the process. Note that the accessing token serves as a pointer since it contains a color and destination field value as shown in figure 9.

```
<accessing token> ::=
                    <color>
                    <destination>
<color>            ::=
                    < {color of
                      stored array
                      values} >
<destination>     ::=
                    < {control input
                      point of storage
                      node} >
```

Figure 9 - Components of an accessing token

STORE is implemented using a COLLECT macro. An array is first represented as a token stream. The eos token is used to determine the array size and is then held on the storage node with its index set to one. Each stream token has its index incremented to reflect its location and is then placed on the SCD node. The COLLECT macro then gathers all the array elements so that they will not be deferred. Bowen discusses several ways of implementing this macro, but if an Increment/Defer matching function is used then the array elements may arrive in any order. The COLLECT macro then clears the tokens' indices and sends them with the ID matching function to a Set-Index node which sets the indices to correspond to the number of tokens that have reached the node thus far. Finally, a signal token is output indicating the arrival of the entire array.

In MAD streams can be declared as being stored, in which case a single token represents a multi-dimensional structure. This leads to the issue of garbage collection, since stored arrays are kept in the Matching Store once they are created.

A modification of the familiar reference count method used for garbage collection on the M.I.T. Data Flow Machine is used. Bowen asserts that it is inefficient to keep a count of the total number of pointers to a structure because reference counts are needed only to indicate that a structure is being shared. Instead of decrementing an array's reference count after it is used, a token would be returned signaling that the use of the array has terminated. In addition, to permit storage sharing, the reference count of a structure would be incremented if it were used in creating a larger structure.

On the Manchester Machine, the SCD node which holds the array tokens also holds the reference count token, whose index is set to zero so it can be distinguished from the array tokens. The count is incremented or decremented using special Increment/Defer and Decrement/Defer Matching functions. An ACCESS macro is used to access a storage node given a pointer to the storage node and a matching function.

A re-entrant garbage collection algorithm is used. First, the array's reference count is decremented and tested to see if it was one before the decrement took place. If it was, no references to the array exist and it can be removed from the storage node. In the case of multi-dimensional arrays, its sub-arrays also need their reference counts decremented. There is a STREAM DECrement macro which accomplishes this. This macro destroys non-pointer input tokens and clears the index field of the sub-array before a pointer can be sent to the start of the garbage collection routine.

Recursive Data Structures

The STORE and SELECT primitives are used to implement recursive data structures. The recursive structure is mapped onto a token stream and is passed to a STORE node. The STORE node outputs a token whose value is a pointer to the structure in memory. Recursive structures are specified using the struct (structure type) notation, which defines the structure by enumerating its generators, which are functions that create a value of the type being defined. The first element of the token stream has an integer value (tag) which tells which generator was used to create the recursive structure. To make the structure recursive the generators have parameters of the same type as the structure being defined (see figure 10). The first token in the stream representing the structure is an integer value which is matched with all possible generators to tell which

one was used to create the structure. The case construct is used to perform this matching. After a generator is selected, a result and signal of completion are output.

```

type node =
  stored struct
    nod(integer,node,node)
    nil {generator with no
          arguments}
  endstruct;

```

Figure 10 - A Recursive Data Structure

It is also possible to have streams of streams. They are created like ordinary streams except that in using the "return all s" statement, s would be a stream itself.

Proposed Extensions

It has been explained that redefining or "updating" one or more elements of an array requires that the entire array be copied. This is because in keeping with the single-assignment philosophy there is no such thing as updating an element that has already been defined. Since arrays are a very useful and conceptually clear way of storing information, and because their applications are so widespread, it is important to investigate better methods for updating them.

Bowen proposes a selective updating construct which has the form:

```
as x with i := <expression>
```

The components of the array x remain the same, except for the ith value which has the value of the evaluated expression. If the array x is not being shared then the i'th array value could be changed in place. For this to be implemented successfully even when the array is being shared, updating cannot be done until all the accesses have terminated. However, the efficiency gained by doing this would be lost if a reference count had to be checked until it became one. For this reason, Bowen suggests that storage sharing and reference counts be eliminated from the model altogether.

Instead, some type of control signaling could be used to announce when all accesses to the array have completed. This would be the only time the array could be updated in place. Otherwise, copying would be necessary. Yet, there is a major problem with this too, because many times an array needs to be accessed after a value has been altered, and this restriction on sequencing will lead to deadlock [6].

Another suggested improvement is to extend streams of streams so that a pointer to the stream could be used before the first element of the

stream has been created. A much higher degree of parallelism could be achieved, but this would require a structure controller that could handle the problem of receiving requests for elements of a stream that have not yet been produced [6].

Finally, Bowen concludes that an extension to the architecture itself would probably be the best approach. Specifically, the Matching Store would have to be changed. Tokens with the same label and destination would be able to enter the Matching Store without an error resulting. When the structure they needed to access entered the Matching Store, it would be required to handle the pending requests for read access to the structure. This extension would increase the difficulty of the job of the Matching Store because an associative match on many values would be called for.

I-STRUCTURES

I-Structures were proposed by Arvind and Thomas [5] as a solution to the problem of excessive copying and slow access time exhibited by normal data structure handling on dataflow machines. An I-Structure is similar to an array but has restrictions on how it is produced and consumed [4]. Storage for I-Structures is allocated before they are produced. Each element of an I-Structure is allowed to be defined (written into) only once, permitting several writes on one I-Structure to take place concurrently. The difference between I-Structures and stored streams is that an element of a stored stream can be accessed only after the whole stream has been created [5].

Since it is possible for a program to read an element before it exists, every element has a presence bit which tells whether that element has been created or not. The attempted read operation is deferred until the corresponding element is created. A complication arises, however, because the deferred reads are queued and once the element is created these deferred reads must be executed. The queues can become very large and checking for deferred reads must be done before every write operation. Thus, the use of I-Structures could become extremely complex.

COMPARISON OF THE M.I.T., UTAH, AND MANCHESTER MODELS

The following statement reveals a paradox inherent in the dataflow models which underlies many of the problems encountered in data structure handling.

The Data Flow Machine offers several benefits for executing determinate algorithms. Many of these benefits are made possible by the absence of side-effects from the Data Flow Language. The lack of side-effects is also the biggest

problem of the language [12].

The designers of the three dataflow models presented have found it necessary to incorporate the concept of memory and the use of pointers into their data structuring schemes. However, the use of pointers has not been explicit, meaning that a recursive type declaration, if it is allowed, would not involve a pointer to the structure type being declared by and directly accessible to the high level language programmer.

Dataflow single-assignment languages cannot support the use of explicit pointers because pointers are used to produce side-effects, whereas the purpose of the single-assignment language is just the opposite. The use of explicit pointers, however, can make selective updating much more efficient. Also, cyclic structures (e.g. circular lists) can be created [6]. Bowen points out that it is difficult to deal with cyclic structures when using a single-assignment language because a pointer cannot be used to move from one point in a structure to another, as can be done with explicit pointers. Rather, it is easiest to view a structure as being an element by itself.

As a result, the acyclic tree structure has proven to be the most suitable representation for complex data structures thus far. Tree representations are used by the Utah DDM1 and the M.I.T DFM. The Manchester machine uses a tree structure for representing dynamic data structures (using the struct notation).

Because the M.I.T. Machine uses a reference count method to detect shared structures and to determine if they can be sent to the free storage list, recursive data structures are not permitted. Re-entrant graphs cannot be used either, so the M.I.T. DFM does not support recursion at all. This is a drawback of the Dataflow nets. Recursion is a powerful characteristic of a language, and fits naturally into most functional programming language frameworks, which are also being explored for data driven machines.

Recursion allows more complex data structures to be defined and gives added flexibility to a language. Also, data structures are often recursive themselves. For example, a tree consists of a root node which is connected to subtrees, each of which may be comprised of other subtrees. As we have seen, the Utah DDM1 and the Manchester Machine do support recursion and recursive data structures as well, although there are still problems with efficiency in these implementations.

On the Manchester Machine, not all data structures are represented by pointer tokens, so the mechanisms used seem to take advantage of pure data flow as much as possible before using the concept of memory and pointers. Actually, on the

Manchester Machine the notion of storage is simulated by special matching functions, as explained earlier. This does not necessarily mean that the use of storage cannot be made efficient enough. The Storage Model of the Utah DDM1, however, seems to be a very flexible storage structure, which is necessary in supporting the variable length list structures and the DDN's. To a large degree, operations of the storage model have been built into the low level hardware functions of the Atomic Storage Unit. There are also disadvantages, however, because flexibility is lost in the fixed, hard-wire tree structure. For example, the processor-store elements (PSE's) in one subtree may have to remain idle if there is a heavily loaded subtree that needs more resources. Also, if one PSE fails, a subtree below that PSE cannot be used [7].

PROBLEMS WITH THE DATA FLOW MODEL

Several problems with the data flow approach to high speed computation have been exposed [9], although the debate over the potential of data flow techniques continues. Little is known about the performance of data flow machines, yet there is definitely overhead associated with token labeling, data storage, and instruction communication.

One major argument is that data flow architectures, while trying to avoid the "von Neumann bottleneck" between memory and the central processing unit, have introduced other bottlenecks with respect to structure memory which are just as costly. For example, we have already observed that upholding the principle of asynchrony leads to conflicts in array access that must be dealt with. Also, if data is being shared it cannot be accessed in parallel, and an increase in memory accesses may result. Finally, the memory space needed to store arrays and the data flow programs themselves can be very large [9].

Even with the use of I-Structures, elimination of the memory contention problem has not been successful. A request for data is sent in one direction, and the I-Structure value is returned in the other direction, just as the von Neumann model references an address and returns a data value. Also, I-Structures require memory to be allocated before it is used and they must be distributed among processors to help decrease network traffic, but an optimal way of doing this remains a problem [9].

The distributed control of the data flow model may also lead to a decrease in speed. With distributed control, individual data paths are controlled separately and often several instructions are needed in place of one instruction in the von Neumann model.

CONCLUSION

While the dataflow model rules out the concepts of control flow and pointers to memory locations it is evident, after viewing the three models presented, that implementations can make use of such concepts. However, the use of these concepts must be invisible to the programmer who should only be aware of values being used in and produced by various computations.

For example, the MAD single-assignment language certainly reflects the semantics of a data flow machine, yet constructs such as the "as...with" and the "for...use" have been developed to try and eliminate some of the problems that arise when only one element of an array needs to be changed. This concept of selective updating is a violation of the single-assignment rule, but these constructs serve as a kind of replacement for the loss in efficiency and usefulness of arrays.

It is difficult to fully evaluate how the Utah DDM1 and the M.I.T. Data Flow Computer actually handle problems such as accessing shared arrays, or selectively updating them. The languages that are being developed for these machines have not been designed with a primary goal of handling data structures efficiently, as is the case with the MAD language.

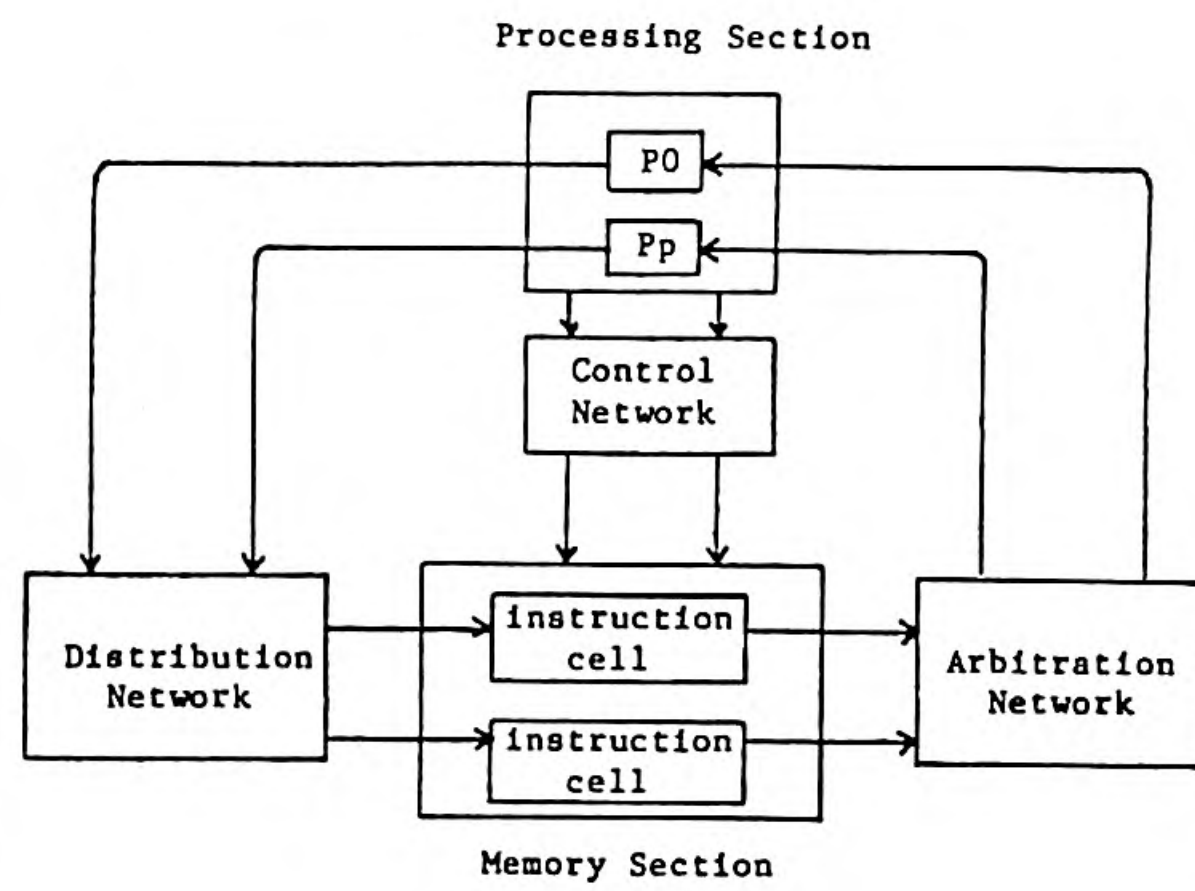
The implementing of data structures on data flow machines remains to be an area that needs much work. Many of the arguments against the data flow model, too, are based on areas which remain highly speculative, and are valid only when certain criteria with respect to such factors as program size and degree of parallelism hold.

Perhaps it is unfair to paint such a bleak future for data flow architectures since high speed architectures in general have many problems that need to be worked out. Gajski compares data flow techniques with optimization techniques which have been known and improved upon for years. It seems more of a justification for trying to implement more efficient garbage collection algorithms and compiler algorithms for controlling memory allocation while upholding the principles of dataflow.

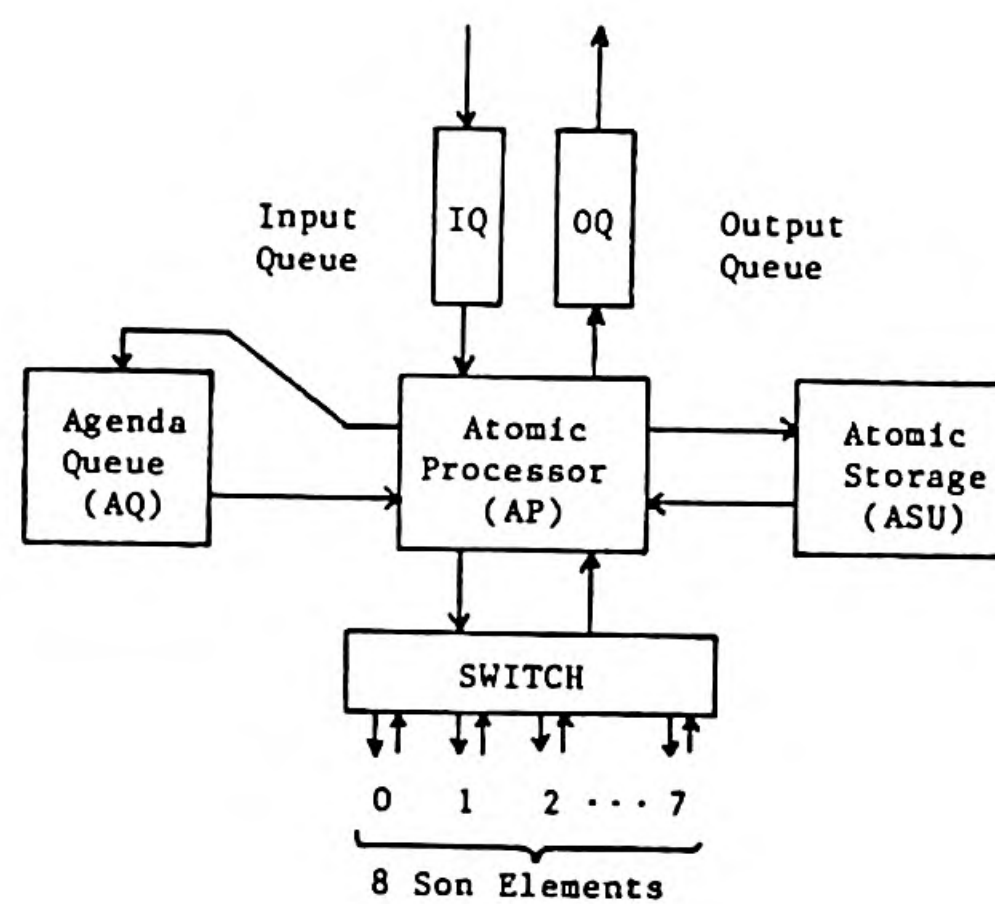
REFERENCES

- [1] W.B. Ackerman, "Data Flow Languages," Proc. of the ACM 1979 National Computer Conference (June 1979), pp. 1087-1095.
- [2] W.B. Ackerman, "A Structure Controller for Data Flow Computers," Laboratory of Computer Science (Computation Structures Group Memo 156), Cambridge, Ma., January 1978.
- [3] W.B. Ackerman, "A Structure Memory for Data Flow Computers," Laboratory of Computer Science (TR-186), Cambridge, Ma., August, 1977.
- [4] Arvind and V. Kathail, "A Multiple Processor Data Flow Machine That Supports Generalized Procedures," Laboratory of Computer Science, Cambridge, Ma., 1981.
- [5] Arvind and R.E. Thomas, "I-Structures: An Efficient Data Type For Functional Languages," Laboratory of Computer Science (TM-178), Cambridge, Ma., September 1980.
- [6] D.L. Bowen, "Implementation of Data Structures on a Dataflow Computer," University of Manchester, April, 1981.
- [7] A.L. Davis and P.J. Drongowski, "Dataflow Computers: A Tutorial and Survey," University of Utah (UUCS- 80-109), Salt Lake City, July 15, 1980.
- [8] J.B. Dennis, "The Varieties of Data Flow Computers," Laboratory of Computer Science, (Computation Structures Group Memo 183-1), Cambridge, Ma., October, 1979.
- [9] D.D. Gajski, "A Second Opinion on Data Flow Machines and Languages," IEEE, February 1982.
- [10] J. Gurd and I. Watson, "A Data Driven System for High Speed Parallel Computing-Part1: Structuring Software for Parallel Execution," University of Manchester, 1980.
- [11] J. Gurd and I. Watson, "A Data Driven System for High Speed Parallel Computing-Part2: Hardware Design," University of Manchester, 1980.
- [12] J.E. Rumbaugh, "A Parallel Asynchronous Computer Architecture for Data Flow Programs," Laboratory for Computer Science (TR-150), Cambridge, Ma., May, 1975.
- [13] P.C. Treleavan, P.R. Brownbridge, and R.P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," Computing Laboratory, University of Newcastle, NE1 7RU, England, 1982.

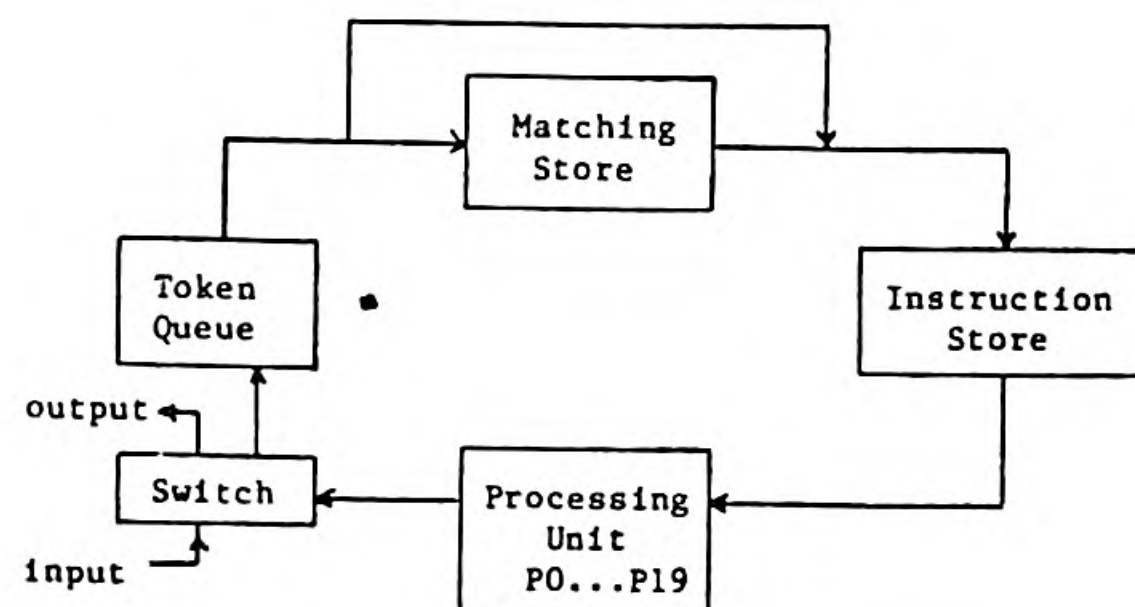
APPENDIX



M.I.T. data-flow computer.



Utah data-driven machine (DDM1).



Manchester data-flow computer.

HARDWARE ARCHITECTURE OF THE PARALLEL FINITE STATE MODEL

A. R. Hurson & B. Shirazi

Department of EECS
University of Oklahoma
Norman, OK 73019

ABSTRACT

The finite state model, its concept, computational power, and applications have been established for many years. Unfortunately, because of the high cost of hardware components in the past, the model has been used as a software tool resulting in a reduced execution efficiency.

The paper introduces a hardware architecture of the finite state model with a high degree of parallelism incorporated into the design. The system is composed of a set of independent units called CELLS. Each cell is a small associative processor, which represents a state and transitions out of that state. In addition, the performance of the system and the gate complexity of each cell have been addressed. These evaluations show that: i) the performance of the system is I/O bound, and ii) the gate complexity of each cell is within the current VLSI technology.

Introduction

Advances in technology during the last decade have had a tremendous impact on the size of the logic circuits, the bit density of the memory, the machine cycle time, and the hardware cost of the computer systems [3,5,15,23]. These improvements brought a greater systems accessibility, the creation of new application areas, and the generation of a new class of users. The nonprofessional computer users have been greatly increased and, as a result, new interfaces between the users and computers have been designed and developed. These changes widened the existing semantic gap between application areas and the hardware of the underlying machine, therefore reducing the efficiency of the computers. This inefficiency is the direct result of mapping the characteristics of the new environments into the characteristics of the conventional computer system.

A high level of efficiency can be achieved by the reduction of the semantic

gap. This goal can be accomplished by: (i) designing the hardware of the computer systems based on the application areas (special purpose computers); or (ii) performing the above mappings directly by hardware (special purpose devices). A comparison of the low cost and high reliability of the hardware devices to the high cost of the software approaches indicates a promising trend in the design and the implementation of more hardware oriented systems.

One of the mathematical models which has been used in many application areas such as text retrieval, compiler design, logic circuit design, control systems, etc., is the finite state automaton (FA). Due to the high cost of hardware in the past, only software simulations of a simpler version of the FA, namely the deterministic model, have been developed and used. This paper presents a hardware implementation of the finite state automaton on an associative memory system. In addition, the paper addresses the time, space, and gate complexity of the proposed architecture. The system is composed of a number of similar processing units called CELLS which are connected via common bus lines. The operations of the cells are independent of each other and they can transfer information and control signals to each other through the common bus lines. Each cell has certain processing capabilities independent from other cells, which allow parallel processing. This parallelism enables us to design and simulate nondeterministic finite state automaton directly by hardware.

The general definition of the finite state automaton, its variations, applications, and implementations will be covered first. Our proposal for the hardware implementation of the finite state automaton and its evaluation will then be discussed.

Definition

A finite state automaton is a mathematical model for any system whose

behavior can be defined by a finite number of states. Formally, a Deterministic Finite State Automaton (DFSA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

Q is a finite set of states,
 Σ is a finite set of input alphabets,
 $q_0 \in Q$ is the distinguished or start state,
 $F \subseteq Q$ is the set of final states, and
 δ is the transition function defined as:

$Q \times \Sigma \rightarrow Q$, that is, $\delta(q, a) \in Q$.

A string $S \in \Sigma^*$ is accepted by a finite state machine if $\delta(q_0, S) \in F$. If δ is modified to allow zero, one, or more transitions out of a state on an input symbol (i.e.: $Q \times \Sigma \rightarrow 2^Q$, where 2^Q is the power set of Q), the nondeterministic (parallel) version of the finite state model is defined.

Applications

Finite state model, based on a strong mathematical foundation, defines the behavior of a vast and diverse set of systems. The control unit of a computer, the computer itself, and even the human brain are only a few examples. Because of the one to one mapping between regular languages (regular expressions) and finite state machines, such a machine can be used to generate the sentences of regular languages, or validate the membership of a sentence in a defined language (regular). Due to the naturalness of the finite state machines in efficient string processing and because of the generality of the model, they have been heavily used in many application areas in computer organizations. In this section an application of finite state model will be discussed:

A SCANNER is an interface between the source file and the rest of the compiler. It is called by the lexical analyzer to produce the next token. A scanner passes over the input program in order to localize and recognize the reserve words, operators, identifiers, labels, etc., used in a particular language. A finite state machine can be used for such a recognition since the production rules for the tokens in a programming language can be defined by regular expressions¹⁶.

Table 1 shows the formal definition of identifiers, integers, comments, strings, and reserve words of the query language ASL¹⁷. Figure 1 depicts the finite state machine for the production rules and part of the reserve words

TABLE 1: PRODUCTION RULES FOR THE TOKENS IN THE ASL.

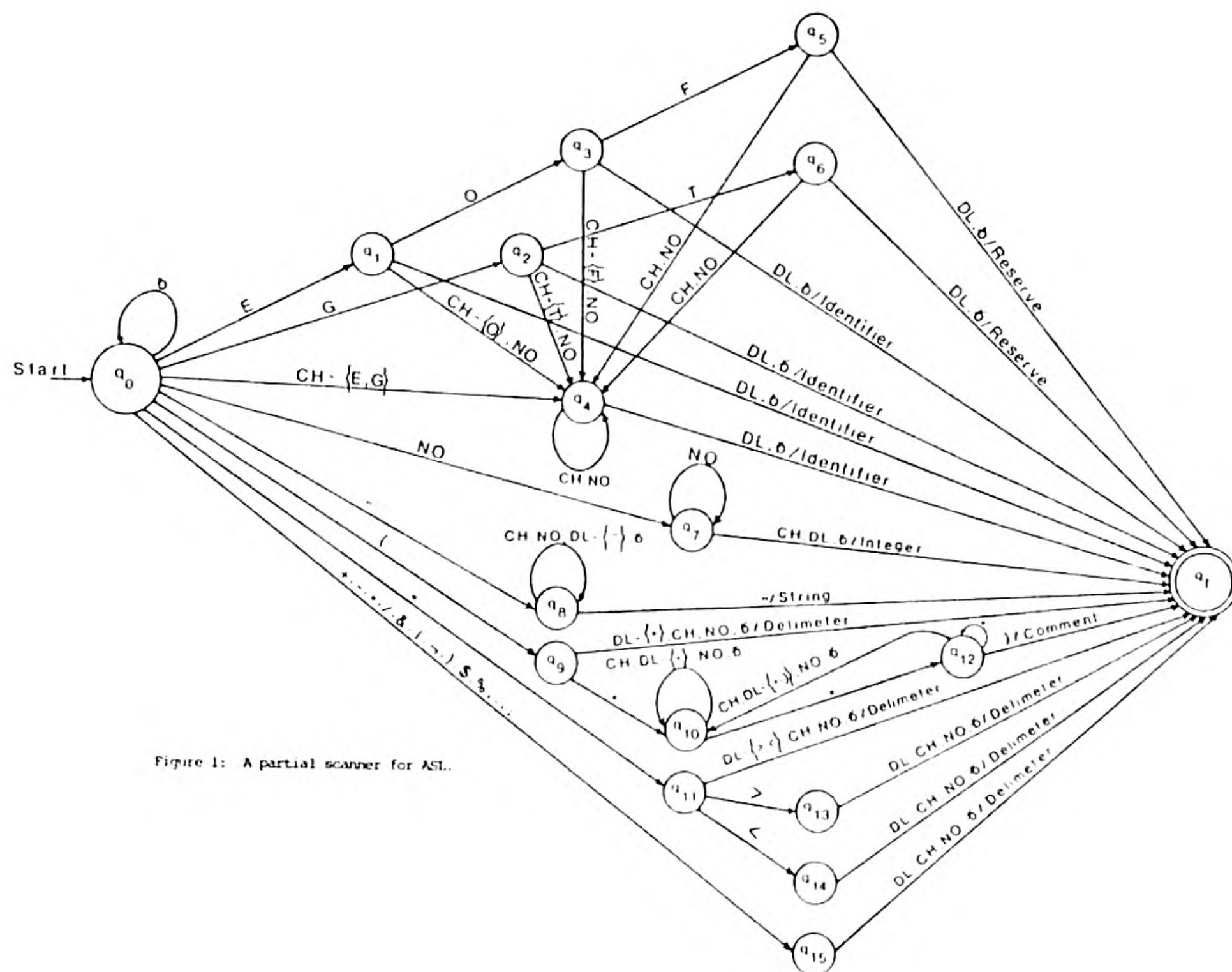
```

<CH> ::= A|B|C|...|Z
<NO> ::= 0|1|...|9
<DL> ::= *|/|+|-|@|||-.|.|>|<|=|<(|)|;|'|b
<IDENTIFIER> ::= <CH>|
               <IDENTIFIER> <CH>|
               <IDENTIFIER> <NO>
<INTEGER> ::= <NO>|
               <INTEGER> <NO>
<STRING> ::= "<ST>"
<ST> ::= <CH>|
        <NO>|
        <DL>|
        <ST> <NO>|
        <ST> <CH>|
        <ST> <DL>
<COMMENT> ::= (*<ST> *)
<RESERVE> ::= BOOLEAN|EQ|GE|GT|INTEGER|LE|LT|NE|
             STRING|EOF|TRUE|FALSE

```

defined in Table 1. To recognize a token, the scanner simply simulates the transition table. Whenever the scanner is called, it starts from the start state and finds a token based on the transition table and the input characters.

In practice, two approaches have been taken toward the simulation and implementation of a finite state automaton: i) software approaches; and ii) hardware approaches²⁵. In both approaches the simulation starts by defining the required parameters in the formal definition of the model. Therefore, the major difference between the two approaches is the way the transition table is represented and searched. In the software approach, a look-up software algorithm searches the transition table based on the current state and input symbol. Such an implementation suffers from two major drawbacks: 1) any software solution is the target of time and space inefficiency, they are error prone and hard to maintain which, in general, reduces the reliability of the system, and 2) the conventional systems are sequential devices and can only handle sequential operations. The parallelism power and efficiency of any parallel operation will be eliminated by the software implementation of that operation on a sequential system. Although parallelism does not increase the computational power of the finite state model, in many instances, such a power can increase the efficiency of the operations and the system. Therefore, conventional



systems are not suitable tools for the simulation of a parallel finite state automaton.

Recently, because of advances in technology, some dedicated systems have been designed and implemented based on the concept of the finite state model^{9,20,24,25}. These systems are able to perform search operations on unformatted data files. Naturally, the hardware implementation increases the efficiency and the reliability of the systems. However, these systems are dedicated to the text retrieval and pattern matching operations, using a large number of states (about 1000)²⁶. Therefore, they may not be cost effective for applications requiring a small to moderate number of states (less than 100). In practice, there are a large number of such applications (e.g. scanner in a compiler). Another disadvantage of existing systems is that they are mostly based on the deterministic FSA. As a result, they do not take advantage of the parallelism embedded in the nondeterministic FSA. The concept of partitioned finite state automata (PSFA)²⁵ allows a limited amount of parallelism.

The Proposed Architecture

In this section we will present an architecture for the deterministic finite state model. The system will then be modified to cover a nondeterministic model.

Deterministic Configuration

In our architecture, a finite state model is composed of a group of identical units called CELLS (i.e. they have the same processing power and storage capacity.) Each cell represents a state and its transitions. The storage holds the transitions out of each cell and the processor performs a search through the contents of the storage according to the input character. The search will be performed in associative fashion on the contents of the storage^{18,22,27}. Each entry in the associative memory has two elements: an input character and a state. An entry represents an arc in the transition diagram of a finite state automaton.

Upon the entrance of the next input character, an active cell performs a parallel search through its contents. In the case of a match, the appropriate row will be marked and the next state will be activated by sending its identity number via the common state bus (Figure 2, a and b). Based on our configuration, a cell could be in one of the three cycles: Busy, Active, or Passive. In the busy cycle, a cell performs a search operation on its contents based on the input character. In a successful search, the next state is determined and will be dedicated for the next input symbol. The active cycle is the duration of time from when a cell is selected as the candidate

state to the time it receives the next input character. Therefore, an active cycle is followed by a busy cycle and during the busy cycle a cell can activate another cell. If a cell is not in the active or the busy cycle, it is in the passive cycle. The duration of an active cycle is a function of: i) the speed at which an input character can be read in; ii) the speed at which the input character can be distributed among the cells; and iii) the time in which the next candidate cell can be activated. Concurrency among operations in steps (i,ii) and step (iii) could reduce the duration of this cycle. The duration of the busy cycle is the function of the associative search processing in the cell, which by itself is a function of the depth of the associative memory, the length of the input character, and the technology on which the associative memory is based.

Figure 2(a) depicts the internal organization of a cell. Each cell is composed of two units:

i) The associate memory: which performs the search operation. The mask register restricts the search to the character field of the rows. Each row of the associative memory can be defined as:

$$\gamma_i \begin{cases} \in (\Sigma \times S) & \text{for } i = 0, 1, \dots, d, \\ = \emptyset & \end{cases}$$

where d is the depth of the associative memory, Σ is the set of the input alphabet and S is the set of the states. In a busy cell, the associative memory is searched against the input character. In a successful search, the second part of the selected row, namely the state field, will be sent to the common state bus in order to activate the next state.

ii) The comparator and match circuit: A unique identity number is assigned to each cell. This unit compares the contents of the common state bus against the cell's identity number. A match indicates that the cell is the candidate for receiving the next input character, otherwise the cell remains in its passive mode. The active signal from the comparator and match circuit activates the search in the associative memory.

Cells can communicate via two common buses: i) the common character bus and ii) the common state bus. Figure 2(b) shows the overall organization of the system. The common character bus is the interface between the outside world and the comparand register of each cell. The common state bus is the communication line between the cells. Cells can activate each other via this connection line which

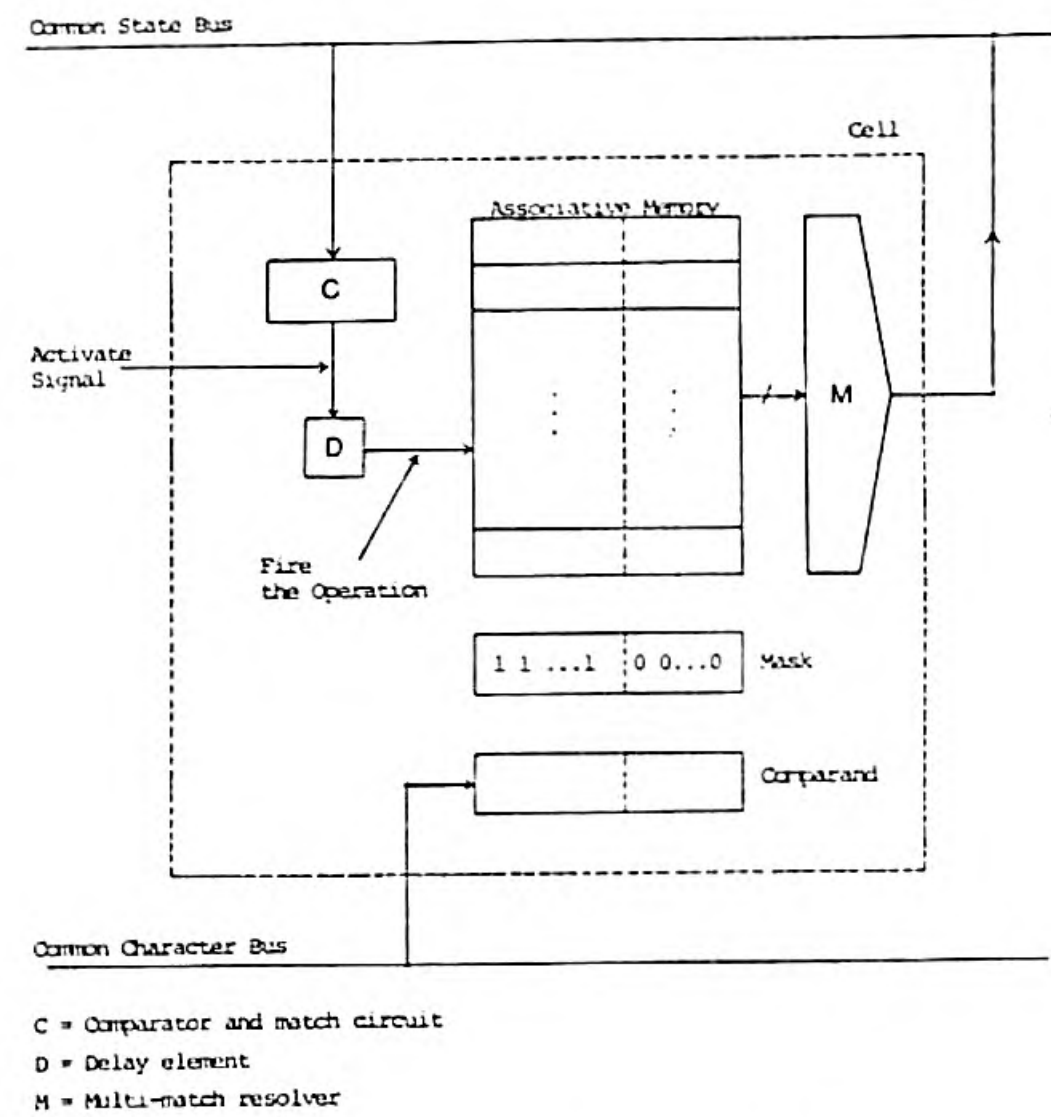


Figure 2a: Block diagram of a cell in the deterministic architecture.

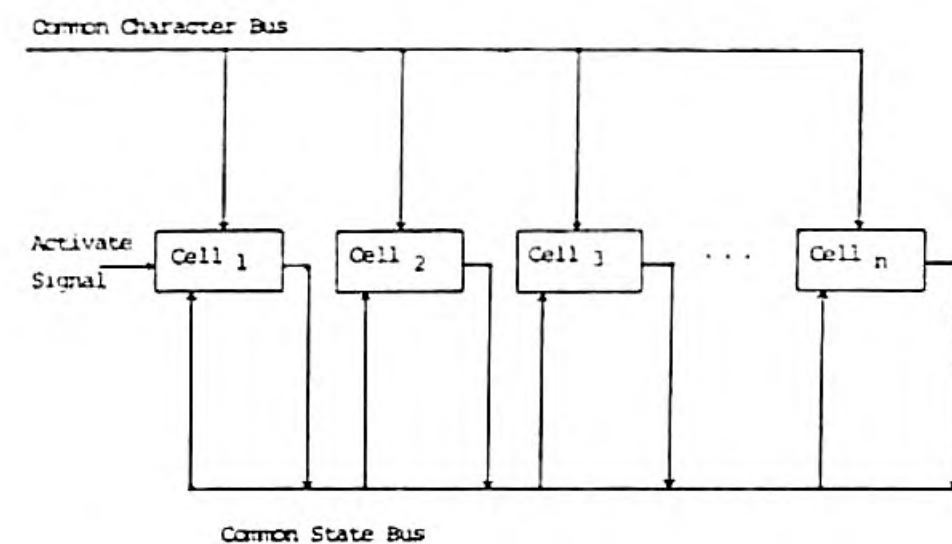


Figure 2b: Overall architecture of the system.

carries the identity numbers associated with the cells. The common state bus can be shared among all the cells without any synchronization of the control over this connection line. This is the result of the deterministic operations, where at any given time no more than one state is in the busy cycle, and at most one cell can be activated in each transition.

Nondeterministic Configuration

While the deterministic finite state model is conceptually simpler than the nondeterministic model, the relatively high number of states and the lack of parallelism are its disadvantages. Such problems can be eliminated in a nondeterministic finite state automaton (NFA), where, there may be zero, one, or more transitions out of a state for an input character. In other words, if $M = (Q, \Sigma, \delta, q_0, F)$ and $\delta(q, a) = p$, then $p \subseteq Q$. This implies that, at each moment of time, more than one state might be in an active cycle.

The implementation of an NFSA on the conventional systems implies a complete copy of the transition table whenever a transition to multiple states occurs¹⁰. This would increase both the time and space complexity of the operations. A modified version of a deterministic finite state model, with a limited amount of parallelism, has been introduced²⁵. This model converts a DFSA into a partitioned finite state automata (PFSA) with the aid of a software algorithm based on a set of heuristics.

The implementation of the nondeterministic FSA, while adopting our proposed architecture for the DFSA, implies the following considerations:

i) Access control over the common state bus: A multi-match requires a busy cell to activate more than one cell as a candidate state. Therefore, the access control of a busy cell over the common state bus is no longer fixed, it is a function of the cardinality of P .

ii) Synchronization of the accesses to the common state bus: Several cells may be in the busy cycle at the same time, requesting access to the common state bus simultaneously. Therefore, the access to the bus must be synchronized. A concept such as daisy chaining can be used to regulate the access control over the common state bus.

An incoming input character may be processed by several activated cells in parallel. The daisy chain mechanism then serially allows each of these busy cells to activate the next set of candidate cells. Therefore, during this time interval, no new character should be read in. This means the read operation should be synchronized with the access control over the common state bus. One solution is to change the delay element in each cell to that of the worst case. If there are (n) cells and the time to activate one cell is (t) , the worst case delay time will be (tn^2) . This delay can turn out to be significant if (n) becomes large. A more elegant alternative is to remove the delay elements and use an external control signal. This signal will be initiated by an external device when there are no busy cells available in the system. Such an external control signal has two functions: i) allowing the flow of the next input character, and ii) changing the state of the active cells to busy.

Figure 3 depicts the organization of a cell for a nondeterministic architecture. The multi-match resolver (M) activates the candidate cells. The last matched row resets the status of the cell to passive. The delay element allows sufficient time

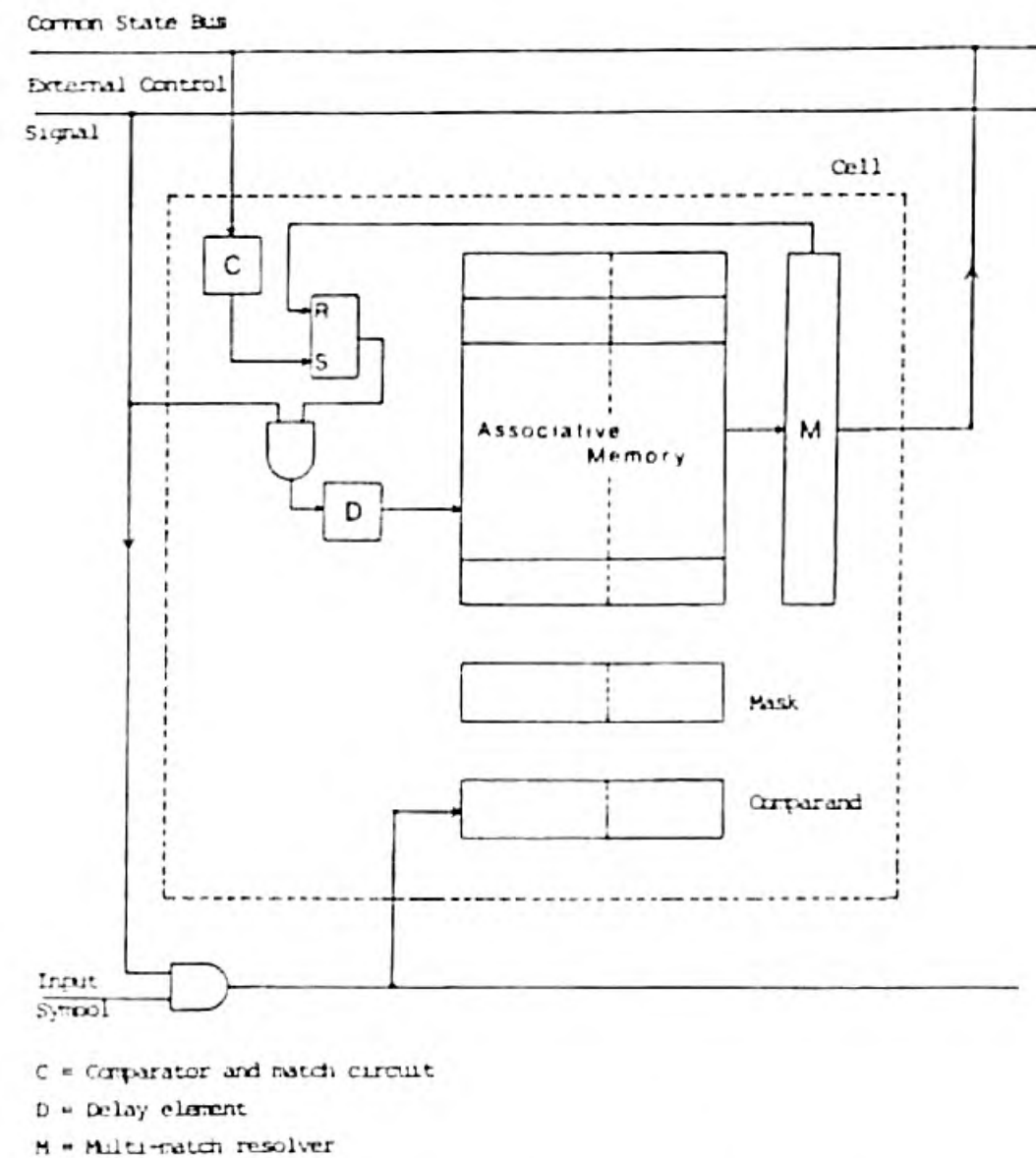


Figure 3: Block diagram of a cell in the nondeterministic architecture.

to fill the comparand with the incoming character.

In the preceding architecture, we can talk about two levels of parallelism: i) the parallel search operation in a cell, which is embedded in the concept of the associative memory, and ii) the parallel operations among several busy cells. Nevertheless, the performance of the system is worse than the performance of the deterministic model by an order of magnitude because the common state bus acts as a bottleneck. However, the performance can be improved by a parallel activation process. This goal can be achieved by providing direct connection lines among all the cells. The scheme can be mapped into a complete graph, where cells are the nodes and the connection lines are the arcs of the graph. For (n) cells, $(n(n-1))$ connection lines are required.

In this system, each busy cell can directly activate other cells through the connection lines. Furthermore, all the busy cells can activate their candidate cells simultaneously. We can, therefore, eliminate the comparator and match unit, the access control mechanism, and the external control signal line. A decoder will be associated to the multi-match resolver of each associative memory. The input to the decoder is the contents of the state part of one of the matched rows, as decided by the multi-match resolver. The output of it will be an activation signal through the connection line corresponding to that state. The activation process time is a function of the number of cells in the system (n) . If

(e) is the execution time of the operations in the multi-match resolver and the decoder, then the worst case time will be (en) .

This scheme offers simplicity and efficiency in comparison to the previous nondeterministic architecture. However, because of the restriction which is imposed by the technology (connection lines) the number of states will be restricted. The problem can be reduced by restricting the number of cells through a concept such as tree-structured finite state automaton.

Because of the above problems, a modified architecture for the NFSA will be proposed. This design offers time efficiency within the limit of the current technology. The new design has the same general organization of the previous models. In addition, a Global Vector Register (GVR) of length (n) , where (n) is the number of cells, is added to the system containing one bit for each cell. Each bit of the GVR indicates whether or not the corresponding cell is a candidate cell for the next cycle. The cells have the same organization as before, except each cell $C(i)$, $1 \leq i \leq n$, is augmented by a Local Vector Register (LVR) (R_i) of Length (n) .

At the end of the busy cycle, each active cell generates a bit pattern in its LVR which represents the candidate cells for the next cycle:

$$R_i(j) = \begin{cases} 1 & \text{If the } j^{\text{th}} \text{ cell is a candidate cell} \\ \emptyset & \text{Otherwise} \end{cases}$$

The R_i 's ($1 \leq i \leq n$) are then used to set the Global Vector Register which is defined as:

$$\text{GVR} = \bigvee_{i=1}^n R_i, \text{ if cell } C_i \text{ is active.}$$

The sequence of the operations is as follows:

1. For each input character, all active cells will perform a search.
2. Each active cell C_i generates the contents of its R_i register. This will be done by the multi-match resolver of each cell.
3. The contents of the R_i registers of all active cells will be ORed and stored in the GVR.
4. The content of the GVR is used to activate the candidate cells for the next input character.

Figure 4(a,b) depicts each cell and the overall organization of the system respectively. The operations in the active cells will be performed in parallel and cells are able to activate the candidate cells without any direct connection lines between each pair of the cells. The number of input/output (I/O) lines of each cell is within the bound of the current technology:

No. of I/O lines = $(m + 1 + n)$
 where m defines the number of bits per character, n is the number of the cells, and 1 is the result of step 4.

For large values of n , the technology could affect the execution time of the step 3 of the above algorithm. In this case the operations of the step 3 will be performed in a partial parallel fashion.

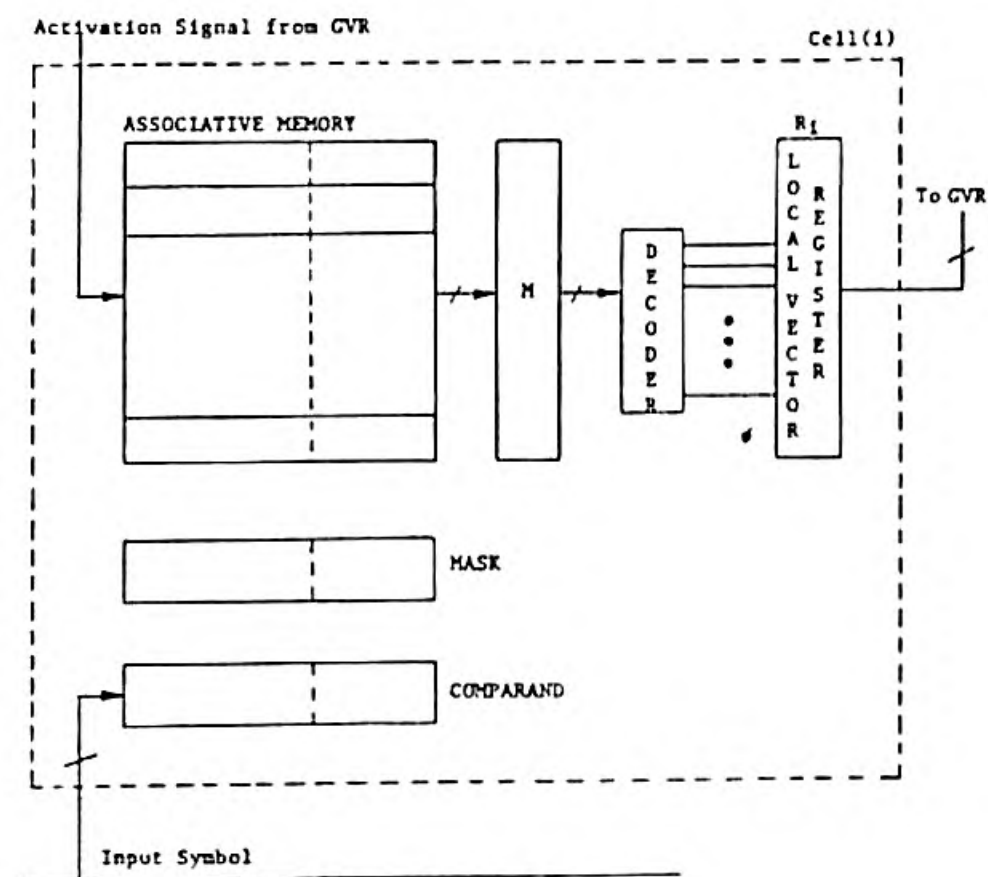


Figure 4a : The architecture of a cell for the Parallel FSA.

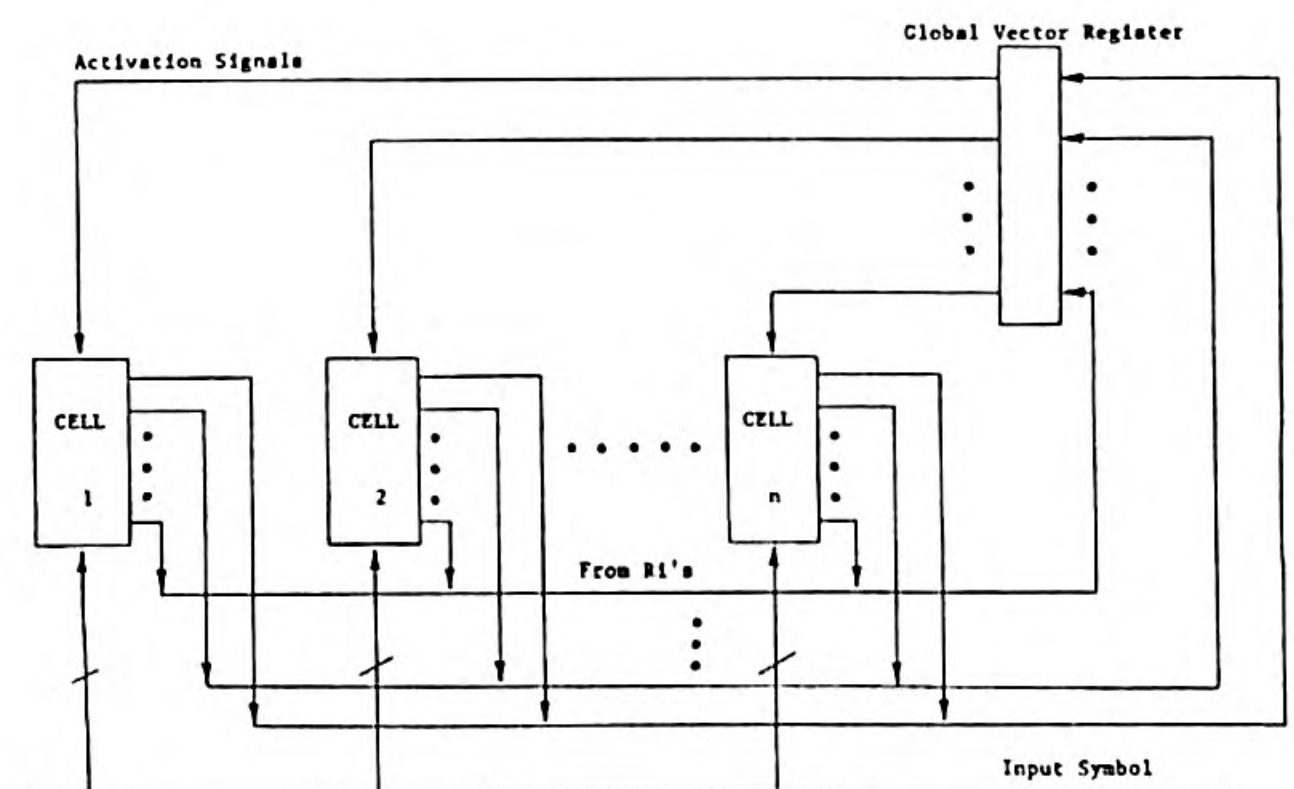


Figure 4b : The overall organization of the Parallel FSA.

Evaluation

In this section the performance of the proposed models will be analytically evaluated. This evaluation is based on the current technology and the flow of operations in each design. In addition, the gate complexity and the number of input/output connections of each model will be investigated.

Table 2 shows the execution time of the different architectures, each consisting of 64 cells (each cell containing 64 words). The evaluation has been based on the following assumptions and criteria:

1) The length of each token is estimated to be eight characters, which is an average English word length.

2) It is assumed that in the NFSA models, on the average, each cell activates 2 candidate cells. This means for the last character of each token, 64 candidate cells will be activated. In reality, some of these candidate cells are common, and hence the total number of candidate cells is generally less than 64.

3) Based on current technology (IBM 4341, STARAN and SPERRY UNIVAC 1100/80), the data transfer time between two registers is estimated to be .2 μ sec.

4) The execution time of the associative search operation is a linear function of the word size. If t is the delay time of a state of logic and C is the character size, the total delay time for setting the match/mismatch bit associated to each word would be:²⁷

$$3\Delta t + [(C-1) + 2]\Delta t = (C+4)\Delta t.$$

In our scheme, where C is 8, the above execution time would be less than .02 μ sec.

5) Allowing a maximum fan-in and fan-out of eight (which is reasonable based on current technology), and using a tree-structured multi-match resolver²⁸, the time to resolve the candidate cells and set the local vector register (R_i) is:

$$(K(2 \log_4 m + 2) + 3)\Delta t$$

where, k is the number of matches and m is the depth of the associative memory.

In addition, if n is the total number of cells, a total of $(4 + \lceil \log_8 n \rceil)\Delta t$ additional delay is necessary in order to set the Global vector registers and activate the candidate cells.

Because of the simplicity of the search operation in each cell (search on equality), each bit of the associative memory requires only four additional logic gates. In addition, since the search will be performed with respect to the input character, the overhead for the gate complexity in each cell would be: $4Cm$, where C and m are character size and the

TABLE 2: The execution time of different systems.

C Character Size.
 n Number of Cells.

SYSTEM	EXECUTION TIME FOR EACH TOKEN (μ sec)	NUMBER OF I/O CONNECTIONS
Deterministic FSA	2.08	$C + 2\log_2 n$
Non-deterministic FSA	103.240	$C + 2\log_2 n$
Non-deterministic FSA (direct connection)	4.04	$C + 2n$
Parallel FSA	.44	$C + n + 1$

depth of the associative memory, respectively. Based on our previous parameters ($C=8$ and $m=64$), the above overhead is about 2K. The gate complexity of multi-match resolver²⁸ and each bit of storage has resulted in a gate complexity of less than 4K for each cell, which is within the range of the current technology^{6,15,19}.

Conclusion

The hardware architecture of the FSA has been discussed. In addition, we have shown how such a scheme can be modified to include the parallelism. The deficiencies of the proposed model with respect to the execution time of the operations and number of input/output connections for each cell were discussed. Such discussion has led us to the design of a parallel finite state model with a fast execution time and a number of input/output connections within the range of current technology. We have shown also that the gate complexity of each cell is in the range of current VLSI technology.

Based on current technology, the latency of the fastest I/O devices is 5 μ sec.²⁹, while the execution time of the proposed system is about .5 μ sec. This simply implies that our proposed system is I/O bound and hence will not create any serious overhead on the performance of the system. The proposed system can be used in the design of the "hardware compiler"¹¹. Such an idea may prove to have a direct effect on the efficiency of the so called "language oriented computers" as well as the conventional von-Neumann systems.

Bibliography

1. Aho, A.V. and Corasick, M.J. "Efficient String Matching: An Aid to Bibliographic Search," Communication of the ACM, Vol. 18, No. 6, June 1975, pp. 333-340.

2. Aho, A.V. and Ullman, J.D. Principles of Compiler Design, Addison-Wesley, 1977.
3. Bloch, E. and Galage, D. "Component Progress: Its Effect on High-Speed Computer Architecture and Machine Organization," IEEE Computer, April 1978, pp. 65-75.
4. Bullen, Jr., R.H. and Millen, J.K. "Microtext-the Design of a Microprogrammed Finite State Search Machine for Full Text Retrieval," AFIPS, 1972, pp. 479-488.
5. Caswell, H.L. et al. "Basic Technology," IEEE Computer, September, 1978, pp. 10-14.
6. "Content Addressable RAM Nears Production," Electronics, February 24, 1982, pp. 63.
7. Gimpel, J.F. "A Theory of Discrete Patterns and their Implementation in SNOBOL4," Communication of the ACM, Vol. 16, No. 2, February 1973, pp. 91-100.
8. Hill, F.J. and Peterson, G.R. Introduction to Switching Theory, Languages, and Computation, Addison-Wesley 1979.
9. Hollaar, L.A. "Text Retrieval Computers," IEEE Computer, March 1979, pp. 40-50.
10. Hopcroft, J.E. and Ullman, J.D. Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.
11. Hurson, A. and Shirazi, B. "Hardware Compilers and Their Impact on the Performance of the Language Oriented Systems." In preparation, 1984.
12. Johnson, W.L. et al. "Automatic Generation of Efficient Lexical Processors Using Finite State Techniques," Communication of the ACM, Vol. 11, No. 12, December 1968, pp. 805-813.
13. Kohavi, Z. Switching and Finite Automata Theory, Second Edition, McGraw Hill, 1978.
14. Knuth, D.E., Morris, J.H. and Pratt, V.R. "Fast Pattern Matching in Strings," SIAM Journal on Computing 6 (no.2), June 1977, pp. 240-267.
15. Lamb, S. "An Add-In Recognition Memory for S-100 Bus Microcomputers," Computer Design, September 1978, pp. 162-168.
16. Lewis II, P.M., Rosenkratz, D.J. and Stearns, R.E., Compiler Design Theory, Addison-Wesley, 1978.
17. Mukhopadhyay, A. and Hurson, A. "An Associative Search Language-ASL for Data Management," AFIPS 1979, pp. 727-732.
18. Parhami, B. "Associative Memories and Processors: An Overview and Selected Bibliography," Proc. IEEE, Vol. 61, June 1973, p. 722-730.
19. Rice, R. Tutorial VLSI - The Coming Revolution in Applications and Design, IEEE Computer Society 1980.
20. Roberts, D.C. "A Specialized Computer Architecture for Text Retrieval," Fourth Workshop on Computer Architecture for Non-numeric Processing, August 1978, pp. 51-59.
21. Thompson, K. "Regular Expression Search Algorithms," Communication of the ACM, Vol. 11, No. 6, June 1968, pp. 419-422.
22. Yau, S.S. and Fung, H.S. "Associative Processor Architecture-A Survey," ACM Computing Surveys, Vol. 9, No. 1, March 1977, pp. 3-28.
23. Martin, R.R. and Frankel, N.D. "Electronic Disks in the 1980s" Computer 8, February 1975, pp. 24-30.
24. Bird, R.M., Tu, J.C., and Worthy, R.M. "Associative/Parallel Processors for Searching Very Large Textual Databases," Third Workshop on Computer Architecture for Non-numeric Processor, 1977, pp. 8-16.
25. Haskin, R.L., "Hardware for Searching Very Large Text Databases," Fifth Workshop on Computer Architecture for Non-numeric Processing, 1980, pp. 8-16.
26. Haskin, R.L. and Hollaar, L.A., "Operational Characteristics of a Hardware Based Pattern Matcher," ACM Transactions on Database Systems, Vol. 8, No. 1, March 1983, pp. 15-40.
27. Foster, C.C., Content Addressable Parallel Processors, Van Nostrand Reinhold, 1976.
28. Anderson, G.A., "Multiple Match Resolvers: A New Design Method," IEEE Transactions on Computers, December 1974, pp. 1317-1320.
29. Kobayashi, H. Modeling and Analysis: An Introduction to System Performance Methodology, Addison Wesley, 1981.

Evaluation of the Manchester Dataflow Machine

Maya Gokhale
Lance Ramshaw
Ioli Constantinou

University of Delaware

The performance of the Manchester Dataflow Machine is studied and discussed. Results of running dataflow implementations of algorithms for parallel search and for the Fast Fourier Transform (FFT) are analysed. The two algorithms were coded in the high level Manchester Dataflow Language (MaD), and statistics were gathered using an emulator for the Manchester machine. Results indicate that a significant degree of parallelism can be achieved in both cases, in spite of the limitations due to inherently sequential portions of the algorithms and to particular features of the high level language.

1. INTRODUCTION

Dataflow has been suggested for almost a decade as a solution to the problem of expressing (as a programmer) and detecting (as an optimizing compiler) all levels of parallelism in an algorithm. A dataflow machine executes a graph which is a representation of the data dependencies in the program, instead of the program counter controlled, sequential list of instructions executed by a conventional machine. The directed arcs of the graph indicate the flow of tokens (data values), while the nodes indicate the execution of an operation [1] [2]. Since nodes can be executed whenever their input data is available, many nodes can usually be executed simultaneously. Indeed, parallelism in a program can be exploited in this way at all levels, from individual expressions up to program blocks.

Although many architectures have been proposed for dataflow machines, to date only a few have actually been built [3] [4] [5] [6]. This paper studies the performance of one such machine, the Manchester University dataflow machine, by examining dataflow implementations of two specific algorithms, parallel search and the FFT. The algorithms were coded in the high level dataflow language Manchester Dataflow (MaD), and the assembly code produced by the MaD compiler was then run on an emulator for the Manchester machine. This paper describes the issues encountered in implementing the algorithms for dataflow, presents and comments on the the performance statistics reported by the emulator for the Manchester machine, and presents an assessment of the dataflow approach for parallel processing.

The two algorithms selected for implementation in MaD, search and the FFT, are of fundamental importance to computer science.

Many of the most difficult classes of problems that computer science faces are exactly those for which no more efficient algorithm is known than a somewhat blind search through the space of all the possible solutions. The combinatorics of such problems seems to dictate that the fastest possible algorithms to solve them must still take time that grows as an exponential function of the size of the input. Thus rapidly solving larger and larger instances of such problems requires a very large commitment of computing resources, and dataflow parallelism may be one way to achieve this. Furthermore, search through a very large space of possible solutions has many applications. One application area of great interest currently is logic programming. Inference machines, which depend on search strategies for unification, have been offered as the computational model for fifth generation computers [7] [8] [9] [10].

The second algorithm, the FFT, is of seminal importance to digital signal processing. Besides being important in its own right, it demonstrates techniques which are used in many other signal processing algorithms.

Dataflow has been recommended as the solution to both these application areas (as well as many others). Considerations in algorithm construction for dataflow and observed performance of the algorithms on the machine should greatly increase our knowledge of the capabilities of dataflow machines.

The following section is a brief description of the Manchester machine, the MaD language, and the emulator. Next follow descriptions of the two chosen algorithms along with the results of running the MaD programs on the emulator. In conclusion, we discuss the implications of programming for a dataflow machine.

2. The Manchester Machine

2.1. Machine Architecture

The architecture of the Manchester machine is diagrammed in Figure 1. The machine consists of a series of functional units connected in a ring topology. Tokens (data) enter the ring from a switch to the host. The tokens are buffered temporarily in a token queue. They next travel the matching store, where tokens destined for the same instruction wait until all such tokens have arrived. Each token is uniquely identified by a label. Tokens belonging to the same instruction have the same label.

Once the token set for an instruction is complete, the set travels to the node store, where all the instructions (nodes) of the program are kept. The token set and the instruction become a group packet, which then goes to a processing element. The instruction is performed on the data, and a result is sent to the switch. The result may be sent to the host as output, or may be recirculated through the token queue as input to a further instruction. The current Manchester prototype has 15 processing elements [11]. This process is described in greater detail in [4].

An important benefit of token labeling is that the dataflow program is inherently reentrant and recursive. Since each token set has a unique label, multiple token sets associated with a single node may be processed concurrently. This implies that processing on all the elements of a stream (array) may occur in parallel. Other dataflow machines are not reentrant, so that elements of a stream must be pipelined to be processed: the instruction may not be shared.

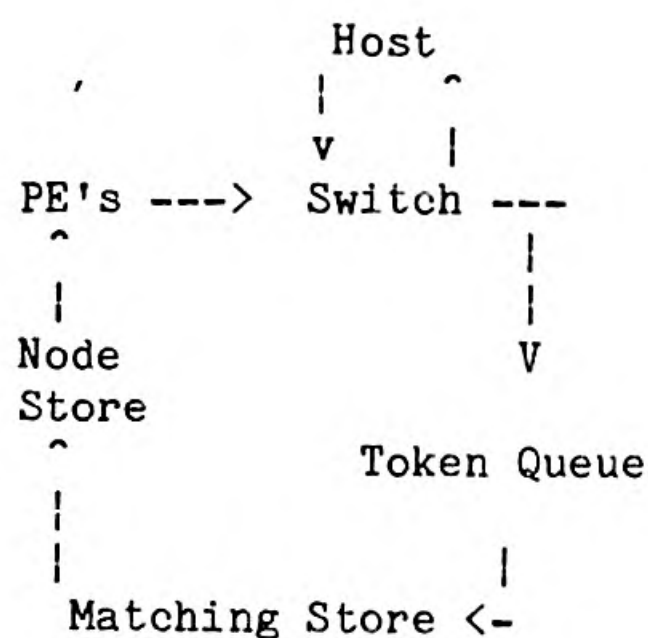


Figure 1 The Manchester Dataflow Machine

2.2. The MaD Language

The MaD language is a high level single assignment language for the Manchester machine [12]. Single assignment refers to the property that a variable can, except under a few very restricted circumstances, be defined only once. Rather than a variable denoting a storage location in memory, a MaD variable is simply a name attached to an expression. Since an expression can have only one value, the variable which denotes the expression can be defined only once.

The MaD language supports data structures such as records and two types of streams—token streams and stored streams. When input to a node consists of a token stream, each member of the stream (the entire array) is input to the node. This can be extremely costly if only one element of the stream is being accessed. Use of a stored stream alleviates the problem. When a stored stream is input to a node, only one token travels to the node. This is a "context" token, or pointer, to the entire stream, which is stored in the matching store. When an element of the stream is accessed from the matching store, the value of the element is copied to the output—the element remains in the matching store. Judicious usage of stored streams can relieve the problem of a large number of tokens circulating in the ring when only one or two are needed by a node. However, there may be a penalty in using a stored stream. Elements of a token stream are available incrementally as produced. Only when all the elements of a stored stream have been generated, however, is any single element available. Therefore, potentially concurrent operations may be forced to run sequentially by use of the stored stream. This effect is demonstrated in one version of the parallel search program.

In addition to the data structuring facilities, MaD has constructs to support iteration (the while construct) and parallel loops. Within a while block, variables local to the block may be reassigned. However, reassignment requires the variable to have a qualifier "new". For example, in a while block controlled by the variable I, the assignment statement

```
new I := I + 1;
```

is permissible. However, a reference to I further textually in the block still refers to I's value before the increment. The value denoted by I is updated to the value of "new I" only after the current iteration is complete. The reader is referred to [13] for a more detailed description of controlled reassignment in iteration blocks.

A parallel loop is invoked in MaD with the for each construct. The form of the loop allows for concurrent processing of loop instances for each member of one or more streams. Since each stream member is identified by a unique label, it

is possible for the machine to process multiple elements concurrently without confusion. The iteration and parallel loop constructs are illustrated below.

2.3. The Manchester Emulator

In the dataflow model, a node can fire at any time after its operands are all available. Parallelism in the Manchester machine comes about when two conditions are met: first, multiple processors are available to fire more than one node at a time; and second, more than one node is ready to fire. The machine emulator works on a series of discrete time steps. At the beginning of each time step, it checks to see which nodes have both their operands, and thus could fire. If there were only one processor available, only one node could fire at a time, and the total number of time steps required to finish the program would be equal to the number of node firings. The number is recorded as S1, the time required for a single processor.

On the other hand, if an unlimited number of processors were available, then all the nodes that are able to fire at a given time step could be fired together, with each getting its own processor. The total number of time steps required to finish the program under these conditions is known as Sinf (the speed with an infinite number of processors). Sinf is a measure of the sum of the lengths of the critical paths through the algorithm, where each step depends in some way on the output of the previous one.

The ratio $S1 / Sinf$ might be called the "potential parallelism" of the algorithm. This label is true in the sense that it shows the speedup factor that could be achieved if an unlimited number of processors were available compared to the time taken by a single processor. This number is only meaningful, however, if the approach used in the algorithm is one that would make sense on a single processor. In some cases, in programming for a dataflow machine, methods must be used which would be startlingly inefficient when viewed from the perspective of a single processor. In such cases, S1 is artificially inflated, and that makes the "potential parallelism" look unrealistically high. Indeed, one could achieve any desired amount of "potential parallelism" just by adding useless extra computation that is unrelated to the rest of the algorithm, which would raise S1 without affecting Sinf. The most important question always must be how fast the parallel implementation is, and how many processors it needs to achieve that speed.

In addition to the S1 and Sinf summary statistics, the emulator provides profiles of the processor usage and matching store usage that occurred while the program was running. These profiles can be divided into at most 100 periods, with each period including the data for one or

more instruction time steps on the Sinf clock.

3. Parallel Search

3.1. A Schema for Searching

While search algorithms are found in widely varying forms in the different areas of computer science, there are three phases, generation, selection, and testing, that are part of every search algorithm, and that help to identify the common features of these varying forms of search. In the generation phase, new elements are added to the search space, either through input or computation. Selection refers to the choice of which elements in the search space to explore first, or which to assign to the available processors, if there are more than one of them; the selection phase controls the course of the search through the search space. In the testing phase, selected elements are examined, returning either a binary, success/failure judgment, or a score indicating the desirability of that choice.

Different forms of search embody these three phases in different ways. For example, in a sequential search of a non-sorted relation in a database for the record with a particular key value, the generation phase reduces to the reading in of the relation, selection is strictly sequential, and the test is a binary equality test on the key field.

In the search of a game tree, as in a program that plays chess, each of the three phases is more complex. Here, since the complete game tree for chess is too large to generate in advance and store, generation means taking a given board configuration and computing the set of all possible next moves. In selection, the program could use some preference function to judge which move to explore next, or it could plan to explore every branch to a certain depth, without caring which are explored first. The test here is either the static board-ranking function applied to the leaves of the tree being searched, or the min-max value which is derived for interior nodes from the leaf scores.

Logic programming also involves the search of a solution space tree. Here the generation of new nodes comes about through the unification of a goal with the rules in the logical database. Selection of which nodes to explore is usually derived from the order in which the goals and clauses are listed in the program, thus allowing the programmer to control the course of the search somewhat. Testing here means simply checking whether or not we have reached an assertion that has no antecedents. Thus very different forms of search can still be expressed in terms of this three-part schema.

3.2. The Promise and Peril of Parallelism

How can parallelism speed up the implementation of this schema? By allowing more than one node to be selected for testing or expansion at the same time. With many processors exploring different areas of the search space simultaneously, we may hope to find the solution faster. At the same time, there is the danger of wasting lots of computing power exploring paths that a sequential algorithm could ignore. If a single solution is being sought, a sequential algorithm that happens to select the correct branch to explore first finds the answer much more easily than a parallel algorithm that explores all branches at once. Another source of overhead in the general case of multiprocessors is the additional complexity involved in dividing up the search space and in managing the inter-processor communication. Although there have not been to date results reported on parallel search specifically on dataflow machines, the tradeoffs in parallel search on varied topologies of multiprocessors have been discussed in [14] and [15]. Results obtained by the former seem to indicate that even in best case (in which the optimal solution is chosen first), a parallel search algorithm can achieve a speedup on the order of $k^{1/2}$ with k processors. Minimization of inter-processor communication is an important design goal in algorithms for parallel search outlined in [10].

3.3. Description of the Sample Problem

The sample problem used in these tests is to search a range of integers and return all the prime numbers found there. Note that this choice of example avoids dealing with the question of interprocessor communication, since the primality test on each integer is entirely independent. In addition, since all values are to be returned, there is no need to kill processes whose answers are no longer needed. These choices were made to keep the problem of reasonable size and to allow the study to focus on the question of implementation of search control strategies in the context of the Manchester machine. The primality test domain also allowed the compact representation of each node in the search space as a single integer, so that the search space could be made fairly large without the running time of the emulator becoming unwieldy.

Using a representation of the search space as a stream of integers, four different control structure options for implementing parallel search in MAD were chosen for investigation: (A) the MAD version of the sequential while loop, (B) the special MAD for each construct, (C) head/tail decomposition of the list through a recursive function, and (D) binary decomposition of the list, also recursively.

Three series of tests were run on the four selected approaches to control of parallel

search. The Series 1 programs tried to use the simplest possible approach. For all but the while control structure, direct input of a stream of integers was judged to be the easiest. For the while loop, the values were derived in the loop on the fly. This avoids the problem of accessing stream elements strictly sequentially.

The tests of Series 2 tried to remedy this lack of uniformity by having all the control structures work with a stream of integers that had been created by an independent while loop earlier in the program. Both Series 1 and 2 used the range of integers (30,50). In Series 3, the programs of Series 2 were rerun with the range of integers to be tested increased to (20,60) so that the work load roughly doubled, in order to see if varying the load would change the relative behaviors of the different control strategies.

3.4. Presentation and Interpretation of Results

For each test, results are given as

$$S1 / Sinf = \text{speedup}$$

where $S1$ is the number of processing steps on a single processor, $Sinf$ is the number of steps if an unlimited number of processors were available, and the quotient gives a measure of the amount of parallelism.

3.4.1. Series 1.

A1.	
While	4077 / 389 = 10
B1.	
ForEach	3837 / 215 = 17
C1.	
Head-	7520 / 2102 = 3
tail	
D1.	
Binary	9892 / 360 = 27
list-	
divide	

In the first Series, the goal was to implement each of the control constructs in the simplest and fastest way, which usually meant providing the stream of integers to be checked for primality directly as input to the whole program. However, in the case of the while construct, the simplest implementation was to input the low and high values, and to compute the values to be checked in the while loop itself. Although there is little explicit evidence of parallelism in a sequential while loop, the parallelism factor turns out to be 10. This parallelism is the sort that comes "for free" in a dataflow implementation, since operations are not forced to be sequential unless there

is some dependency. Thus here, the while loop can run ahead generating new items while the earlier calls to the function primetest (which tests a number for prime) are still running.

Program B1 uses the explicitly-parallel for each construct. Thus there is no explicit while loop, and the candidate values, instead of being generated, are fed to the program as an input stream. As one might expect, this is the fastest implementation of all, both in terms of S1 and Sinf, although it is not the "most parallel". In comparing A1 with B1 in Series 1, note that while B1 does almost as much overall work (S1), it is able because of the for each construct to overlap it considerably more.

In program C1, a directly input stream is fed to a function that tests the head item and either adds or does not add it, as appropriate, to the list of successes generated by calling itself recursively on the tail. This form of breaking up the input stream turns out to be more expensive, both in terms of S1 and Sinf, than that of A1 or B1. It also has the distinction of having the lowest parallelism factor of any of the tests run. It is, in fact, mainly useful as a contrast to the next case, D1, in which the stream is divided in half each time, and the function calls itself recursively on each half. This process, computing the midpoint and segmenting the list, involves extra work, so that D1's S1 figure is higher than that for C1. However, it allows for much more concurrency, so that its Sinf figure is second only to that of the for each version. However, it is important to note that D1's speed would depend on having more than 100 processors available; the extra work involved in dividing the stream is only being made up for by unusually broad concurrency.

3.4.2. Series 2

A2.
While (a) $14805 / 666 = 22$
(b) $7241 / 476 = 15$

B2.
ForEach (a) $4583 / 365 = 12$
(b) $5074 / 480 = 10$

C2.
Head- $8545 / 2400 = 3$
tail

D2.
Binary $10918 / 572 = 19$
list-
divide

more comparable for the different control structures, each program was written so that low and high values were the input, and a separate while loop was used at the beginning of each program to compute the stream of candidates.

For the while loop version, this raised the question of how the while loop could get at the values in the stream. In program A2a, this was achieved by indexing into the stream, which MAD allows. However, it is clear that the indexing operation is quite expensive in terms of S1, since it is implemented in an iterative way, although enough of the indexing operations were overlapped with each other so that the Sinf figure is not wildly higher, given the extra work involved. In program A2b, the stream is segmented in the while loop by setting a loop variable each time to the tail of the previous stream, and this turns out to be faster than indexing, both in terms of S1 and Sinf.

Program B2a, based on the for each construction, just shows the extra work compared to B1 of computing the candidate stream rather than inputting it directly. This not only causes extra work, but also reduces the possible parallelism. B2b is just like B2a, except that the stream of candidates is declared stored, which in this case makes things even slower. With a stored stream, the entire stream must be generated before the prime test can be performed on any element.

C2 and D2 show only the expected minor increases in both S1 and Sinf when compared to C1 and D1. The comparative rankings for Series 2 only change from Series 1 in that the tail-resetting form of the while loop version A2b has crept ahead of the binary list-dividing version D2 in terms of Sinf.

3.4.3. Series 3

A3.
While (a) $44138 / 1209 = 36$
(b) $18390 / 810 = 22$

B3.
ForEach (a) $8666 / 579 = 14$
(b) $19737 / 751 = 12$

C3.
Head- $23318 / 5013 = 4$
tail

D3.
Binary $22553 / 854 = 26$
list-
divide

In Series 2, in order to make the results

The third Series of tests, using the same programs as the second Series, examines the effect of doubling the size of the range of integers to be checked for primality. This turned out in most cases to increase the Sinf rating to a little less than twice what it was in Series 2, which is much as one might expect. The comparative standings between the different algorithms remained exactly the same. It is interesting to note, however, the behavior in terms of S1. Here the while loop version that used indexing, A3a, showed a more-than-threelfold increase, a sign of the fact that indexing down longer streams is comparatively more expensive. The head-tail functional version C3 also showed almost a threefold increase in S1, and was the only one to more-than-double its Sinf figure, but it is no surprise that that form of list decomposition does not lend itself to massive parallelism.

3.4.4. Summary of Results

These tests demonstrate that a considerable amount of parallelism in search programs can be achieved fairly easily in a dataflow context. This parallelism can greatly increase the speed of a program's execution, though the amount of speedup depends on the particular control structure implementation of parallel search in the chosen language, as well as on the number of processors and thus the breadth of parallelism available.

In the MAD language, the for each construct proved to be the most efficient of the four tested control structures for implementing simple parallel search of a stream. While loop tail-resetting and recursive binary list-division were almost as efficient in terms of Sinf, but only by doing much more work, spread among many more processors.

4. The Fast Fourier Transform

The FFT (Fast Fourier Transform) is an efficient algorithm to compute the DFT (Discrete Fourier Transform) of a digital signal [16]. For an N-point signal, straightforward calculation of the DFT requires N^2 complex multiplications and $N(N-1)$ complex additions. However, if $N = 2^k$, the version of the FFT described below requires only $N(k/2)$ complex multiplications and Nk complex additions [17].

On a conventional machine, the FFT has an added benefit. Usage of memory can be reduced since computation of dual node pairs can be performed in place. That is, memory locations used at one iteration for a pair of dual nodes can be reused in the next iteration to store the new values of the dual nodes. This feature is of course not useful on a dataflow machine—there is no global updatable storage. However, the dramatic reduction in complex multiplications and additions over the straightforward DFT calculation is motivation enough to produce a dataflow FFT.

In the following sections, we develop the FFT algorithm in terms of a signal flow graph, which is very similar to a dataflow graph. The MaD FFT implementation is then described. We note in this section programming techniques developed which were necessitated by the dataflow environment. The last subsection presents the results of running the program on the Manchester Emulator.

4.1. A Description of the FFT Algorithm

The FFT is an efficient implementation of the DFT, which is given by equation 1 below:

$$X(n) = \sum_{k=0}^{N-1} x_0(k) e^{-j2\pi nk/N}$$

with $n=0,1,\dots,N-1$

eq.{1}

where n describes the frequency domain of a waveform and k describes the time domain. We also use the following conventional notation as presented in [17]:

$$W = e^{-j2\pi/N}$$

where N is the number of equations we have to compute. If N is 4 then:

$$\begin{aligned} X(0) &= x_0(0)W^0 + x_0(1)W^0 + x_0(2)W^0 + x_0(3)W^0 \\ X(1) &= x_0(0)W^0 + x_0(1)W^1 + x_0(2)W^2 + x_0(3)W^3 \\ X(2) &= x_0(0)W^0 + x_0(1)W^2 + x_0(2)W^4 + x_0(3)W^6 \\ X(3) &= x_0(0)W^0 + x_0(1)W^3 + x_0(2)W^6 + x_0(3)W^9 \end{aligned}$$

eq.{2}

and in matrix form:

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} W^0 & W^0 & W^0 & W^0 \\ W^0 & W^1 & W^2 & W^3 \\ W^0 & W^2 & W^4 & W^6 \\ W^0 & W^3 & W^6 & W^9 \end{bmatrix} \begin{bmatrix} x_0(0) \\ x_0(1) \\ x_0(2) \\ x_0(3) \end{bmatrix}$$

eq.{3}

In this experiment the number N was considered to be a power of 2, so $N=2^A$ where $A > 0$, and A is an integer. Equation {3} can be

rewritten as:

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & W^1 & W^2 & W^3 \\ 1 & W^2 & W^0 & W^2 \\ 1 & W^3 & W^2 & W^1 \end{bmatrix} \begin{bmatrix} x_0(0) \\ x_0(1) \\ x_0(2) \\ x_0(3) \end{bmatrix}$$

eq.{4}

since $W^{nk} = W^{nk \bmod N}$

Factoring the matrix and by rearranging some factors:

$$\begin{bmatrix} X(0) \\ X(2) \\ X(1) \\ X(3) \end{bmatrix} = \begin{bmatrix} 1 & W^0 & 0 & 0 \\ 1 & W^2 & 0 & 0 \\ 0 & 0 & 1 & W^1 \\ 0 & 0 & 1 & W^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & W^0 & 0 \\ 0 & 1 & 0 & W^0 \\ 1 & 0 & W^2 & 0 \\ 0 & 1 & 0 & W^2 \end{bmatrix} \begin{bmatrix} x_0(0) \\ x_0(1) \\ x_0(2) \\ x_0(3) \end{bmatrix}$$

eq.{5}

Notice that $X(2)$ and $X(1)$ are interchanged. Then we let:

$$\begin{bmatrix} x_1(0) \\ x_1(2) \\ x_1(1) \\ x_1(3) \end{bmatrix} = \begin{bmatrix} 1 & 0 & W^0 & 0 \\ 0 & 1 & 0 & W^0 \\ 1 & 0 & W^2 & 0 \\ 0 & 1 & 0 & W^2 \end{bmatrix} \begin{bmatrix} x_0(0) \\ x_0(1) \\ x_0(2) \\ x_0(3) \end{bmatrix}$$

eq.{6}

$x_1(k)$ is equal to the product of the last two matrices of equation {5}. The element $x_1(0) = x_0(0) + W^0 x_0(2)$ is computed with one complex multiplication and one complex addition, as is $x_1(1)$. Only one complex addition is needed for $x_1(2)$, since $W^0 = -W^2$ and

$$\begin{aligned} x_1(2) &= x_0(0) + W^2 x_0(2) \\ &= x_0(0) - W^0 x_0(2) \end{aligned}$$

eq.{6-a}

($W^0 x_0(2)$ has already been computed for $x_1(0)$). Also, $x_1(3)$ only needs one complex addition, for the same reason.

Multiplying the matrices of equation {6}, we get:

$$\begin{bmatrix} X(0) \\ X(2) \\ X(1) \\ X(3) \end{bmatrix} = \begin{bmatrix} x_2(0) \\ x_2(2) \\ x_2(1) \\ x_2(3) \end{bmatrix} = \begin{bmatrix} 1 & W^0 & 0 & 0 \\ 1 & W^2 & 0 & 0 \\ 0 & 0 & 1 & W^1 \\ 0 & 0 & 1 & W^3 \end{bmatrix} \begin{bmatrix} x_1(0) \\ x_1(1) \\ x_1(2) \\ x_1(3) \end{bmatrix}$$

eq.{7}

The computation of $x_2(0)$ needs one complex multiplication and one complex addition:

$$x_2(0) = x_1(0) + W^0 x_1(1)$$

and as in the previous case $W^0 = -W^2$, thus $x_2(2)$ requires only one complex addition.

In order to compute equation {3} sixteen complex multiplications would have been needed, and twelve complex additions. But with equation {5} we need only four complex multiplications and eight complex additions. In general the FFT requires $N\log_2 N$ complex multiplications and $N\log_2 N$ complex additions, if $N=2^{\log_2 N}$, compared to the direct method which requires N^2 complex multiplications and $N(N-1)$ complex additions. Thus we get a ratio of $2N/\log_2 N$ direct/FFT computing time. We must not forget though that one extra step has to be performed with the FFT, the rearrangement of our results (see equation {5}). This can be done very easily if we view the indices as binary numbers. Example:

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} \Rightarrow \begin{bmatrix} X(00) \\ X(10) \\ X(01) \\ X(11) \end{bmatrix}$$

eq.{8}

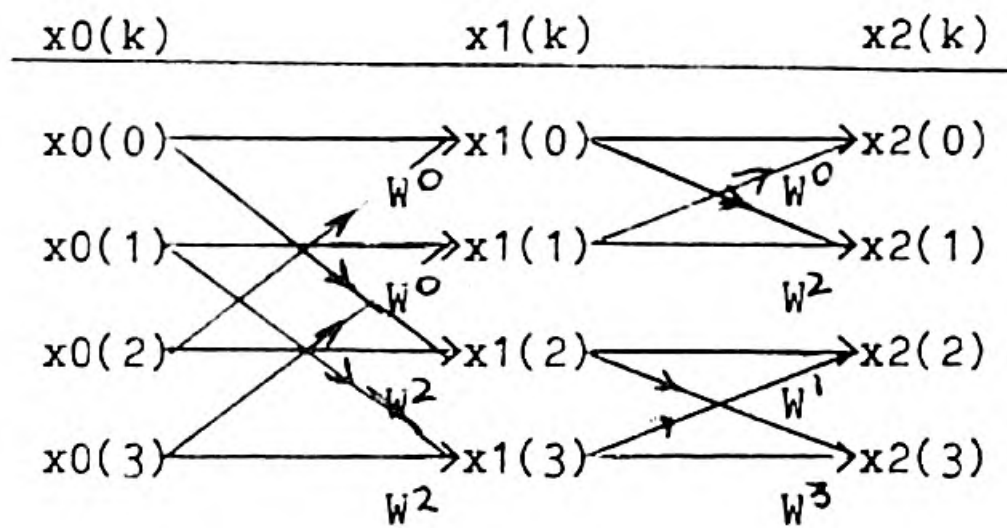
Reversing the indices of equation {8} we get:

$$\begin{bmatrix} X(00) \\ X(10) \\ X(01) \\ X(11) \end{bmatrix} \Rightarrow \begin{bmatrix} X(00) \\ X(01) \\ X(10) \\ X(11) \end{bmatrix}$$

eq.{9}

which has the right order.

The FFT can be also represented as a flow graph where we have computational columns ($N=2^{\gamma}$). For $\gamma=2$ we have:



eq.{10}

The signal flow graph is very similar to a data flow graph. We have paths that carry values to nodes. The values that are carried down on these paths are multiplied by a specific W^p , and the result is then entered in the node of the next array. If there is no W^p appearing on the path the value is assumed to be one. Then the values that enter a node through the paths are added together (two for each node). For example, in order to compute node $x1(2)$ we have:

$$x1(2) = x0(0) + W^2 x0(2)$$

which is equation {6-a}.

Thus for the FFT equation {5}, we can compute each column of the matrix using the signal flow graph method. In the previous example first we would compute $x1(k)$ and then $x2(k)$. Also, from the previous example we can see that certain nodes have exactly the same input as is the case for nodes $x1(0)$ and $x1(2)$. They are both computed with $x0(0)$ and $x0(2)$ as inputs. Thus there is no reason why we should perform the same computations twice. Rather we multiply and add the inputs together once, and store the result in both nodes. These nodes are called dual nodes. The space between two such nodes is found to be $N/2^m$ where $N=2^{\gamma}$ and m is the number of the column we are computing. For $x1(0)$ and $x1(2)$ m would be 1. Only one complex multiplication is involved in the computation of dual nodes, since if a node needs factor $W^{N/2}$, then its dual node needs factor W^p , and $W^p = -W^{N/2}$. So, we compute dual nodes by:

$$\begin{aligned} x_m(k) &= x_{m-1}(k) + W^p x_{m-1}(k+N/2^m) \\ x_m(k+N/2^m) &= x_{m-1}(k) - W^p x_{m-1}(k+N/2^m) \end{aligned}$$

eq.{11}

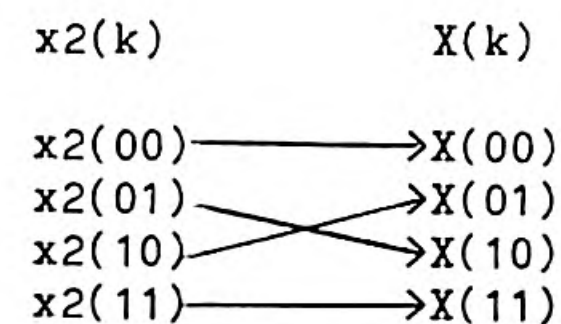
When computing a specific column, we want to skip

over dual nodes which have already been computed. That is, we compute the first $N/2$ nodes and then skip over the next $N/2$ nodes until we reach the bottom of the column, which is $N-1$.

When computing the values for the various nodes we assumed availability of the W^p 's needed for each one of these nodes. The value of p is computed as follows:

- 1) Write the index of the node in binary with γ bits.
- 2) Shifting the binary number by $\gamma-m$ bits to the right and filling in the opened positions with zeros.
- 3) Reverse the order of the binary bits.

In order to get back the results in the correct order (remember eq.{5}), we have to use a method for unscrambling the last column produced. We write the index of each node in binary and then reverse that number, converting back to decimal. That will give us pairs of nodes, which have to be interchanged. As we go down the column interchanging $X(k)$ with $X(i)$, we will at some point encounter a node that has already been interchanged, and in order to avoid interchanging it again we just have to check if the integer i , produced by bit reversing, is less than k , the index of the column. If so, it means that we already unscrambled that particular index. The above is illustrated in the following example:



eq.{12}

4.2. The Dataflow Program

The basis of the FFT is equation {5}. Our goal is, given the $x0(k)$, $0 \leq k \leq N-1$, column as input to our algorithm, to compute all the $xi(k)$'s, $1 \leq i \leq \gamma$, necessary to obtain a value for $X(k)$'s. Then $X(k)$'s are the output of the algorithm. Given an equation that describes a specific waveform for which we seek to find its Fourier Transform, we first have to choose sample points for the equation and then find the numbers for the real part and the imaginary part, if any, of those points. These numbers will then constitute our $x0(k)$'s of eq. {5}. Once we have the input sample points, and the value of our γ , we can compute N (see previous section).

Here the need arises for a function that will create all the columns of our signal flow graph ($xi(k)$'s, $1 \leq i \leq \gamma$, see eq.{10}). Every time we create a new column, as mentioned above, we use

as input the values of the nodes contained in the column just preceding the one being built. For example, $x_2(k)$ will use the values that were generated for $x_1(k)$ and so on. Also this function has to keep track of the space between two dual nodes. Each time a new column is created the dual space decreases by half, and the reason for N being a power of 2 is exactly that. We want the dual space to be divided by a known number for every new column creation. In the last column the dual node space becomes one.

Another necessary function is one that will compute the nodes of the current column, given the values contained in the previous column (see eq. {6}). This function creates a column by means of a stream [12]. In MaD a stream always has to be initialized if we want to add more elements to it within a loop. So, to initialize the stream representing a column we set its first element to the values of the dual node pairs contained within two dual spaces. Then, every iteration cycle of a while loop creates the next set of dual nodes, which are concatenated with the already created values of the column. (Later on we will see in greater detail how the values are handled within cycles of a while loop.) This action has to be performed twice, once on the real part and once on the imaginary part of our points.

The use of streams restricts the possible different methods which one may want to use in order to create one column. The elements of a stream have to be created in sequence. Thus, we could not create a node and its dual, store them in a stream and go on to create another pair. This becomes even more apparent in the next function, responsible for determining the various W 's and also for creating the actual values for the dual pairs. But on the other hand, the return all feature of MaD can be used. The return all allows a stream to be created with values that a 'variable' takes within a while loop (see the new construct discussion above) with the first element in the stream corresponding to the value that the 'variable' had during the first iterative cycle and the last element the value of the last cycle. The return all makes it possible to create and store in a stream all nodes that are contained within the current dual space: that is, all single nodes without their duals. Most important though, is that concurrently we can be creating the duals of the above nodes, which will be stored in another stream. Thus we obtain two streams, one containing single nodes and the other containing their duals in the correct sequence. By concatenating these two streams we get a complete set of dual nodes. The number of these nodes will be equal to $\text{dualspace} * 2$.

This is the method used to create, first, sets of dual nodes, then a whole column and finally all the columns necessary to obtain the last and the most important one. It is worthy to note here that we can use the new construct to

create a new instance not only of a simple variable, but of a whole stream. Therefore the values of a column were replaced by the values of the new column being generated by the use of the new on streams.

Our results are not totally correct yet. We still have to unscramble the last column to get the results. (see eqs. {5}, {8}, and {9}.) Again here we run into the problem of creating elements of a stream in random order. From $x_i(k)$, $i = \gamma$, we have to create $X(k)$. One way to overcome this burden is to create a stream that will contain the order with which the indices will have to be extracted from x_i . With the for each construct we can access all the elements of the above stream sequentially. These elements though, will be indices of another stream x_i , and since we can access elements of a stream in a random order, we can extract the value from the specified index. This is the final function, that unscrambles the stream and produces a new one with the values in the correct order.

The functions described above are given in their full form in [18]. Here we focus on only one function in order to demonstrate the dataflow approach to algorithm construction.

As mentioned above, eq. {12}, the way to obtain the correct order of the results is to convert the indices of the last column, γ , to binary, reverse the binary number and convert back to decimal. Here arises a need for a function that takes an index and finds its corresponding one in the new array according to the above algorithm. This function is presented here:

```
function revbit
  (index,gamma:integer):integer;
  declare
    j1,nextj1,k,j2,i:integer;
  init
    j1:=index;
    i:=1;
    k:=0;
    j2:=index div 2;
    nextj1:=j2;
  while i <= gamma do
    new j2:=nextj1 div 2;
    new k:= k*2+(j1-2*j2);
    new nextj1:=j2 div 2;
    new j1:=nextj1;
    new i:= i + 1
  return k;
```

Figure 2 'revbit' as
a MaD function

For comparison we also provide the corresponding sequential function in Pascal:


```

function revbit
  (index,gamma:integer):integer;
var
  j1,j2,k,i:integer;
begin
  j1:=index;
  k:=0;
  i:=1;
  while i <= gamma do
    begin
      j2:=j1 div 2;
      k:=k*2+(j1-j2*2);
      j1:=j2;
      i:=i+1
    end
  revbit:=k
end;

```

Figure 3 'revbit' as a Pascal function

The sequential function uses fewer variables and fewer steps. The reason for this is the nature of the while loop, the new construct of MaD, and the effect of the parallel execution of statements. Everything inside the while loop (between the 'while' header and the 'return') may be executed in parallel. So, before entering the loop we have to initialize all the 'variables' to the values that we want them to have during the first execution cycle. The word 'variables' is used in quotes since MaD is a single assignment language and does not allow a change in the value of an identifier unless with the use of the reserved word new. So, when we enter the loop the first time j2 will immediately get its 2nd value, but the computation of k will involve the 1st values of j1 and j2. The need for a new variable, nextj1, arises when inside the while loop, since statements are computed in parallel, we need to have two different values for the same variable, j1. This variable is initialized to the value of the index, and in the while loop it gets the value of j2. (see the sequential algorithm.) But j1 is also used to compute k using the value that it has before j2 gets assigned to it. Now, since these two statements can be executed in parallel, we do not know whether j1 will change before, or after it gets the new value of j2. If the programmer is not aware of this, then the wrong value of j1 may be used. Thus nextj1 is used as an intermediate step within cycles, providing an extra value. In the sequential version, the extra variable is not required. We can control the order of reassignment to j1.

With the use of the new construct a variable can be reassigned a value and keep that value during a whole cycle of the loop. The value changes only between cycles. The return k statement will give function revbit the value that k had during

the last iterative cycle.

4.3. Results

The data used in this experiment were points from equations whose FT (Fourier Transform) is known [19]. These equations are as follows:

Original	FT
a. 1 for x<b 0 for x>b	----> (2*sin(b*a))/a
b. $\frac{1}{x^2 + b^2}$	----> $(\frac{e^{-ab}}{b})/b$
c. $\frac{1}{w - b}$	ibx ----> ie returns >0 when x>0 <0 when x<0

Figure 4 The equations used for the FFT program

For (a) the first equation of fig. 4 we set b=3, a=2πn/N, where n is the index of the node. We used 4 sample points, that is, we chose γ to be 2. The values of these points were provided in the data section at the end of the program. Since this function does not have an imaginary part, the 4 numbers corresponding to this part were set to zero.

For (b) the second equation of fig. 4 we set b=1, a=2πn/N and chose γ to be 3, thus we provided 8 numbers for the real part of the sample points and 8 for the imaginary part. The latter were set to zero for the same reason as for the previous equation.

The last equation (c) of fig. 4 has both real and imaginary parts. Four sample points were given, and again b=1, and w=2πn/N.

In contrast with the parallel search programs mentioned above, the nature of the FFT is to a large extent sequential. This was confirmed after examining the processor usage profiles and the storage profiles of the various runs.

While loops were used for the computation of

the columns of the signal flow graph, and for the computation of dual nodes within columns. Parallelism was high within these loops since all their statements could be executed in parallel. Also the computations of the real and the imaginary part of the results did not depend on each other extensively, thus parallelism was also high whenever such computations had to take place. The results of running the three equations are as follows:

- a. $s1 = 16746$
 $\text{sinf} = 432$
 $s1/\text{sinf} = 16746/432 = 38$
- b. $s1 = 55949$
 $\text{sinf} = 628$
 $s1/\text{sinf} = 55949/628 = 89$
- c. $s1 = 16746$
 $\text{sinf} = 628$
 $s1/\text{sinf} = 16746/432 = 38$

Figure 5 Potential Parallelism

The processor profile of the running program shows distinctive peaks and valleys in the amount of parallelism. There are some steps which are independent and can therefore be executed concurrently. Others are strictly sequential in nature. The latter compose a large part of the algorithm. It can be seen from the processor profile that parallelism varies considerably from subprogram to subprogram. For example the computation of the signal flow graph column is highly parallel, whereas the unscrambling of the last column is sequential in nature. The average parallelism of the program is 30%.

Both processor and storage profiles indicate that the degree of parallelism that depends on the values of the input points is very small. Comparing the outputs of two runs where each used 4 points, we can observe that for the most part they are the same, with a few variations. The $s1$ and sinf results indicate that these variations balance out, since both runs end up with the same potential parallelism. We also observe that the parallelism increases more than linearly with the number of input points, since the $s1/\text{sinf}$ for 4 points is 38 and for 8 points is 89. The amount of computation that takes place in an FFT program that is run with a large number of input points is considerably greater than the computation of an FFT program which uses fewer input points. More computation causes the potential parallelism figure to be greater, since certain parts of the program that can be computed in parallel are executed more times. χ will be greater, thus the while loops will perform more iterative cycles, and we

will also have more nodes to compute for each computational column.

Unfortunately the performance results given by the emulator of the Manchester machine do not easily relate to the MaD program statements. The action of tokens during program execution is given in assembly code and the processor and storage profiles refer to this code. Thus, the programmer who seeks to find the bottlenecks within a program, or the statements that increase the parallelism, has to be able to relate the results to the assembly code. A useful performance analysis tool would be one which correlates results from the machine code to the high level language statement.

5. Conclusion

The performance results of the search programs and the FFT indicate that a significant amount of parallelism was achieved in both cases. We have seen that search programs achieve a higher parallelism with the use of the for each construct. A large amount of speedup can be obtained through the parallel implementation of the FFT also. For a complete description of the parallel search programs see [20], and for the FFT see [18]. However, in order to obtain correct results and an efficient implementation, the programmer must understand the dataflow computation model. Dependence on implicit sequential instruction execution can cause mysterious bugs in an asynchronous dataflow environment.

References

- [1] Dennis, Jack B., "First Version of a Dataflow Procedure Language", Lecture Notes in Computer Science, V. 19, 1974, pp. 362-376.
- [2] Gostelow, Kim P., and Thomas, R. E., "Performance of a Simulated Dataflow Computer", IEEE Transactions on Computers, V. C-29, No. 10.
- [3] Sauber, W., ed., "A Dataflow Implementation", Texas Instruments, 1980.
- [4] Watson, I., and Gurd, J., "A Practical Dataflow Computer", IEEE Computer, Feb. 1982.
- [5] Comte, D., Hifdi, N., and Syre, J.C., "The Data Driven Lau Multiprocessor System: Results and Perspectives". IFIP, 1980.
- [6] Davis, A.L., "A Data Flow Evaluation System Based on the Concept of Recursive Locality". IFIP, 1979.

- [7] Moto-oka, T., ed., Fifth Generation Computer Systems, North Holland, JIPDEC, 1982.
- [8] Hogger, C. J., "Concurrent Logic Programming". In K. L. Clark and S.-A. Tarnlund, eds., Logic Programming. Academic Press (1982), 199-211.
- [9] Pereira, L. M., and L. F. Monteiro, "The Semantics of Parallelism and Co-routining in Logic Programming". In B. Domolki and T. Gergely, eds., Mathematical Logic in Computer Science. North-Holland Press (1981), 611-657.
- [10] Van Emden, M. H., and G. J. de Luceno Filho, "Predicate Logic as a Language for Parallel Programming". In K. L. Clark and S.-A. Tarnlund, eds., Logic Programming. Academic Press (1982), 189-198.
- [11] Gurd, John and Watson, Ian, "Data Driven System for High Speed Parallel Computing". Part 1,2. Computer Design, June, July 1980.
- [12] Bowen, D. L., "Implementation of Data Structures on a Data Flow Computer". PhD. Dissertation, Univ. of Manchester, April 1981.
- [13] Ackerman, William B., "Data Flow Languages". NCC, 1979.
- [14] Finkel, Raphael A., and John P. Fishburn, the "Parallelism in Alpha-Beta Search". Artificial Intelligence 19 (1982), 89-106.
- [15] El-Dessouki, Ossama Ibrahim, and Wing H. Huen. "Distributed Enumeration on Network Computers". IEEE Trans. on Computers C-29 (Sept. 1980), 818-825.
- [16] Aho, Hopcroft, Ullman, "The Design and analysis of Computer Algorithms". Addison-Wesley, 1974, 252-276.
- [17] Brigham E. O., "The Fast Fourier Transform". Prentice-Hall, Inc., 1974.
- [18] Constantinou I., "A Data Flow Implementation of the Fast Fourier Transform". Univ. of Delaware, 1983. (CIS unpublished)
- [19] Greenberg M. D., "Foundations of Applied Mathematics". Prentice-Hall, Inc., 1978.
- [20] Ramshaw L. A., "Parallel Search Techniques on the Manchester Dataflow Machine". Univ. of Delaware, 1983. (CIS unpublished)

RECURSIVE 2 DIMENSIONAL MEDIAN FILTERED SIGNALS WITH IMPULSIVE NOISE*

R.J. CRINON AND G.R. ARCE

University of Delaware
Electrical Engineering Department
Newark, Delaware 19711

ABSTRACT

Median filtering is a nonlinear technique which is efficiently used to smooth signals perturbed by heavy tailed distributed noise. In this paper we find the output distribution of recursively median filtered two dimensional signals with additive impulsive noise. The edge jitter generated by the recursive median filtering of a two dimensional signal is then analyzed. The results are compared with those obtained from the standard median filter. Finally some examples are included to illustrate this theoretical results.

1 INTRODUCTION

In many data acquisition systems such as image processing, impulsive type noise is added to the original signal and linear filters are not suitable for eliminating this type of noise. Median filters have proven very useful in several applications: they are easily implemented on digital processors and can do an excellent job in eliminating impulse type noise without deterioration of edges.

There are two types of median filter operations; the *Standard median filter* and the *Recursive median filter*. To date, no comparison has been carried out to determine whether or not one class is more efficient in smoothing noisy two dimensional signals and at the same time tracking the original signal. To this end, two criteria can be distinguished; one is the ability to eliminate the noise; the other, is how much distortion does the filter introduce to important signal structures; structures such as edges, which are important in image processing. In this paper, we compare the separable standard and recursive separable median filter under these two criteria, when additive white impulse noise is present.

- First, we will consider the statistical properties of both filters on constant two dimensional signals plus additive white impulse type noise.
- The second part will analyze the edge jitter introduced by the recursive filtering on an edge corrupted with white additive impulsive type noise. This will be compared to the edge jitter introduced by the standard median filter.
- Finally, some examples are presented to illustrate these theoretical results.

Before starting the description of some of these different parts, a review of the one dimensional standard and recursive median filter is in order.

The standard median filter.

If $\{x(\cdot)\}$ and $\{y(\cdot)\}$ are the input and the output sequences respectively, then the standard median filter output for a window of size $2n+1$ is:

$$y(a) = \text{median}[x(a-n), \dots, x(a), \dots, x(a+n)]. \quad (1)$$

The recursive median filter

The recursive median filter is a modification of the standard filter where the value of the center sample of the window is replaced by the computed median value before the window is shifted to the next position. In other words, the filtered sequence for a filter with window width $2n+1$ is

$$y(a) = \text{median}[y(a-n), \dots, y(a-1), x(a), x(a+1), \dots, x(a+n)]. \quad (2)$$

Gallagher and Wise¹ have shown that at most $(L-2)/2$ passes of the standard median filter are required to reduce an L length signal to its root, which is a smoothed version of the input that is invariant to further median filtering. The advantage of the recursive operation is that the root signal is obtained after the first pass of the recursive filter²; hence reducing processing time. This is desirable if real time processing is needed. Although the roots generated by the standard and recursive filter may differ for the same input signal, they have the same characteristics.

The separable median filter

When filtering two dimensional data, the median filter cannot only change in size but also in shape. Square and cross shaped windows are commonly used; however, it has been observed that these two dimensional windows do not preserve all the edges present in the two dimensional signal. A combination of two one dimensional median filters called the *Separable median filter* has shown to perform well. Separable median filtering is a two pass operation. The first step is the filtering of each row of the input image with a one dimensional horizontally oriented median filter of window width $2n+1$, as seen in Fig. 1. n points are appended to the beginning and end of each row equal to the value of the first and last point respectively to allow the filter to reach the edges of the image. The columns of the resulting filtered signal are then filtered with a vertically oriented one dimensional filter of window width $2n+1$. Recursive or standard operations can be used. More explicitly, the output value $y_{i,j}$ of a recursive separable median filter is given by

* The authors gratefully acknowledge the support of the National Science Foundation under Grant // ECS - 830 7764

$$y_{i,j} = \text{median value of } \left\{ y_{i,j-n}, \dots, y_{i,j-1}, z_{i,j}, \dots, z_{i,j+n} \right\}, \quad (3)$$

where

$$z_{i,j} = \text{median value of } \left\{ z_{i-n,j}, \dots, z_{i-1,j}, x_{i,j}, \dots, x_{i+n,j} \right\}, \quad (4)$$

where the $x_{i,j}$ are the elements of the two dimensional input.

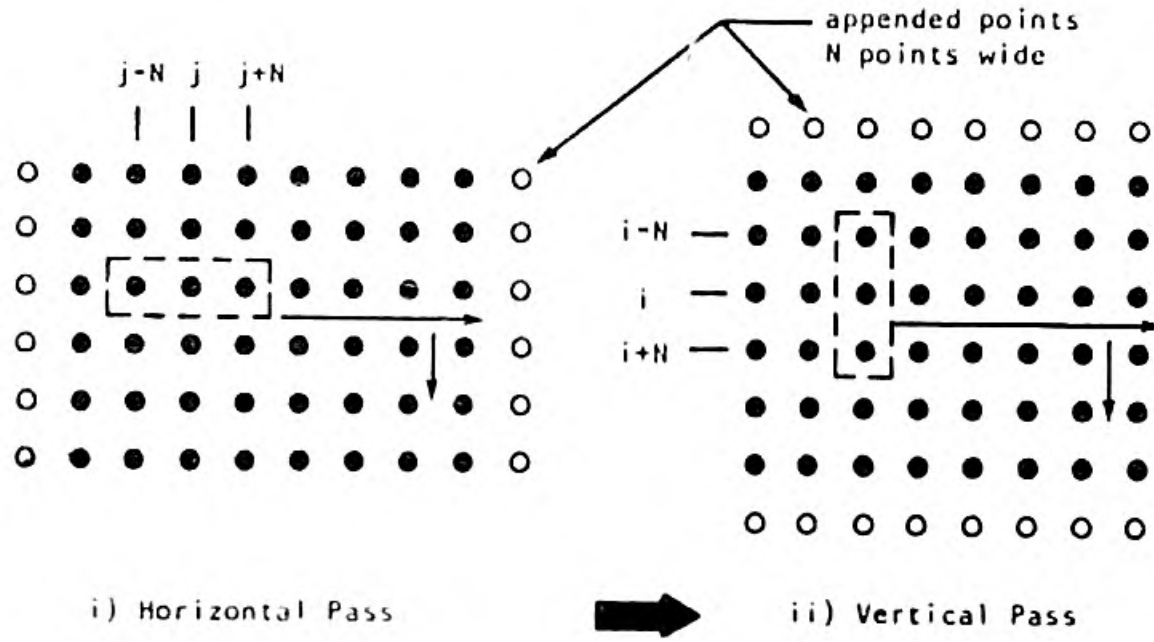


Fig. 1: The separable median filter.

II THE FILTERING OF IMPULSE TYPE NOISE

1) Filtering of positive Impulsive Noise.

In this section, we first derive the output distribution of two dimensional recursively median filtered signals with white, additive impulsive noise (for any window size). These results are then used to compare the efficiency of the standard filter against that of the recursive filter in eliminating the impulsive noise. Since the noise is white, there is no correlation introduced between the two passes of the recursive separable median filter and we can treat this problem as two one dimensional filtering processes. For a one dimensional signal "s(i)",

$$s(i) = c + n(i), \quad (5)$$

where in this case c is the signal (zero) and the noise is distributed as

$$P_r(n) = \begin{cases} \alpha & \text{for } n = 0 \\ \beta & \text{for } n = 1 \end{cases}, \quad (6)$$

the output distribution of the recursive median filtered signal was obtained by Crinon and Arce⁷ and is given by

$$\pi_{1d}(0) = \frac{\alpha^{n+1} + n\beta^{n+1}\alpha^{n+1}}{\alpha^{n+1} + 2n\beta^{n+1}\alpha^{n+1} + \beta^{n+1}} \quad (7)$$

$$\pi_{1d}(1) = \frac{\beta^{n+1} + n\beta^{n+1}\alpha^{n+1}}{\alpha^{n+1} + 2n\beta^{n+1}\alpha^{n+1} + \beta^{n+1}} \quad (8)$$

where $\pi_{1d}(0)$ and $\pi_{1d}(1)$ are the stationary probabilities for a "0" and a "1" respectively, after the first pass of the separable filter. Equation (8) is the probability of an impulse occurring at the output of the first pass of the recursive separable median filter as a function of β and n. Recall that the filtering in one direction does not introduce correlation on the other direction; consequently, the output distribution for the separable recursive median filter signal is obtained by using $\pi_{1d}(0)$ and $\pi_{1d}(1)$ instead of α and β , respectively in equations (7) and (8); hence, the output distribution of the separable recursive median filter is:

$$\pi_{2d}(0) = \frac{\pi_{1d}^{n+1}(0) + n\pi_{1d}^{n+1}(0)\pi_{1d}^{n+1}(1)}{\pi_{1d}^{n+1}(0) + 2n\pi_{1d}^{n+1}(0)\pi_{1d}^{n+1}(1) + \pi_{1d}^{n+1}(1)} \quad (9)$$

$$\pi_{2d}(1) = \frac{\pi_{1d}^{n+1}(1) + n\pi_{1d}^{n+1}(0)\pi_{1d}^{n+1}(1)}{\pi_{1d}^{n+1}(0) + 2n\pi_{1d}^{n+1}(0)\pi_{1d}^{n+1}(1) + \pi_{1d}^{n+1}(1)} \quad (10)$$

Equation (10) is the probability of positive impulses at the output of the recursive separable median filter.

For the standard separable filter, the probability of an impulse at the output of the horizontal pass of the filter is³

$$P_h(1) = \sum_{k=n+1}^{2n+1} \binom{2n+1}{k} \beta^k (1-\beta)^{(2n+1)-k} \quad (11a)$$

and the probability of an impulse at the output of the vertical pass is

$$P_v(1) = \sum_{k=n+1}^{2n+1} \binom{2n+1}{k} P_h(1)^k (1-P_h(1))^{(2n+1)-k}. \quad (11b)$$

In (11a) and (11b), the binomial coefficient is defined as:

$$\binom{2n+1}{k} = \frac{(2n+1)!}{k! (2n+1-k)!} \quad (12)$$

The previous analysis is now used to compare the recursive and standard separable median filters. For example, the probability of an impulse occurring at any signal point is decreased from $6.0 \cdot 10^{-2}$ to 10^{-26} by filtering the signal with a standard separable median filter of window 11. The same result can be obtained by filtering the noisy signal with a recursive separable median filter of window length 9. Figure 2 shows the probability of an impulse occurring at the output of the filtered noisy two dimensional signal, as a function of the probability of an impulse occurring at the input, for both standard and recursive filters.

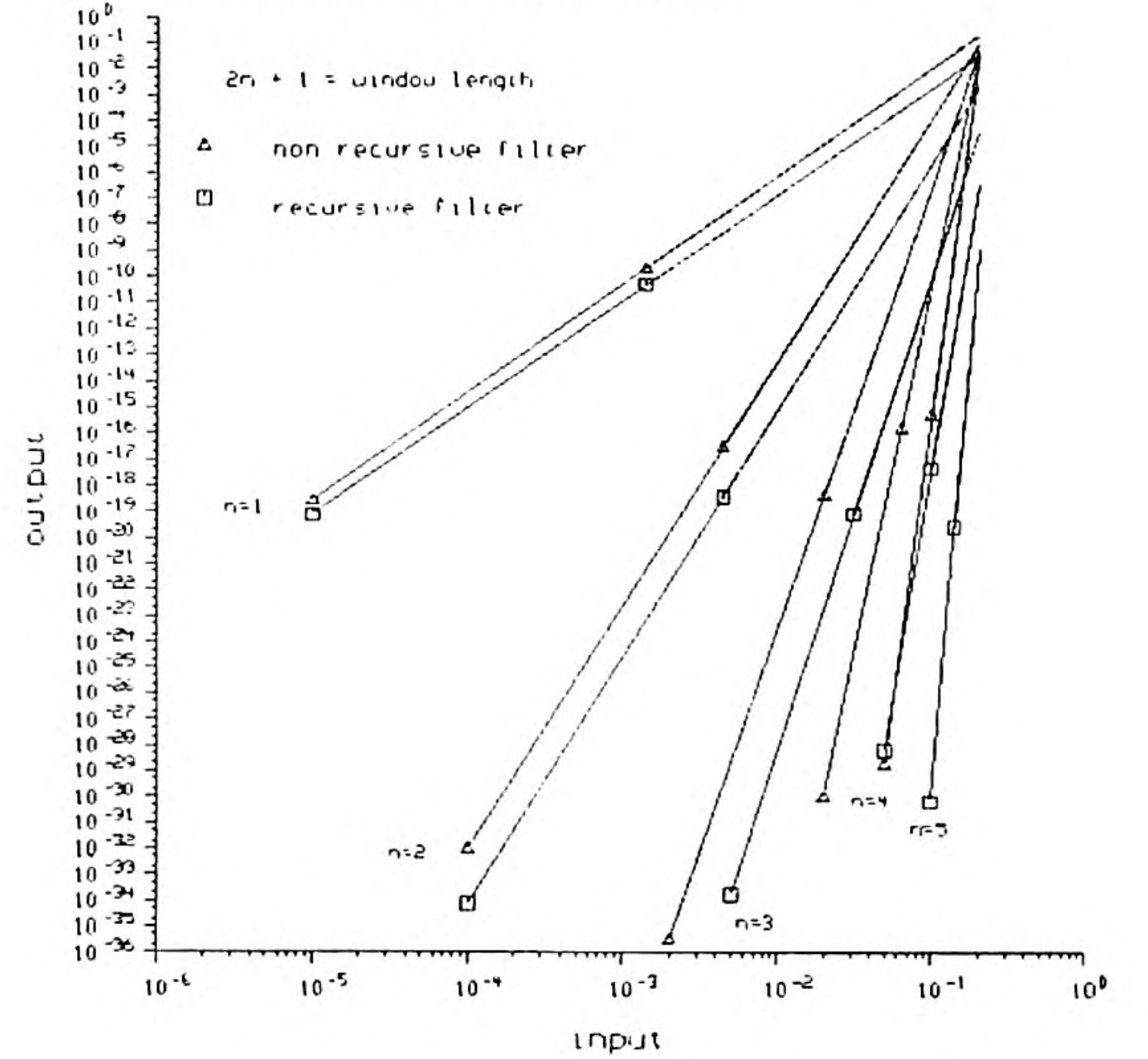


Fig. 2: Probability of positive impulse

2) The filtering of positive and negative impulsive type noise.

Consider the noisy two dimensional signal:

$$s(i,k) = c + n(i,k), \quad (13)$$

where in this case $c = 1$ is the constant signal. The noise is white and distributed as

$$P_r(n) = \begin{cases} \alpha = 1 - 2\beta & \text{for } n = 0 \\ \beta & \text{for } n = 1 \\ \beta & \text{for } n = -1 \end{cases} \quad (14)$$

As it can be noticed from (14), one more level has been added to account for the negative impulses; the existing levels are "0", "1" and "2". For a window length 3 up to 9, the

stationary probabilities of each level have been computed; particularly, the probability of positive or negative impulses at the recursive median filtered output is given by $\pi(0) + \pi(2)$. The results are plotted Fig. 3. Again the performance of the recursive filter is better than the standard filter. Moreover it can be noticed that, given β and n , the probability of an impulse at the output of the recursive median filter is lower in Fig. 3 than in Fig. 2. This is due to the existence of symmetric impulsive levels; an original level in the signal can be also retrieved from a window containing the same number of positive and negative impulses. For example the probability of an impulse (either positive or negative) occurring at any signal point is decreased from $1.5 \cdot 10^{-1}$ to $2.0 \cdot 10^{-10}$ using a standard median filter of window length equal to 9. The same results are obtained with a recursive filter of a window length equal to 7.

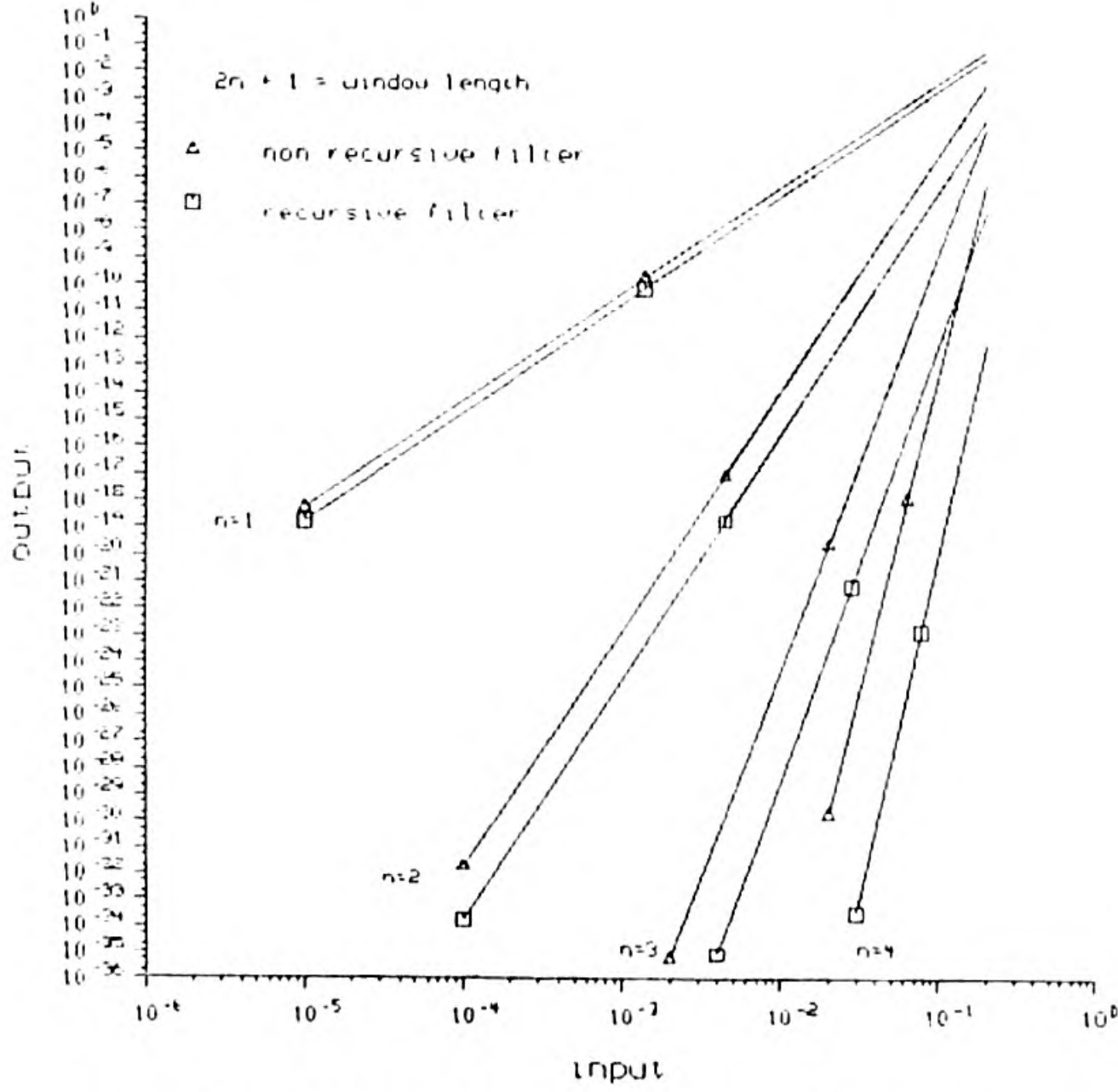


Fig. 3: Probability of positive and negative impulse.

III EDGE JITTER DUE TO IMPULSIVE TYPE NOISE

In this section we analyze the edge jitter introduced by the filtering of a noisy signal. As in the previous section, we begin with the analysis for white positive impulsive type noise. This analysis is then extended for positive and negative impulses.

1) Edge jitter due to positive impulsive type noise.

Consider the two dimensional signal $g(i,j)$ shown in Fig. 4. This signal contains an edge at coordinate $(m+1)$ on the X-axis; we assume that this edge spreads infinitely along the Y-axis. Hence

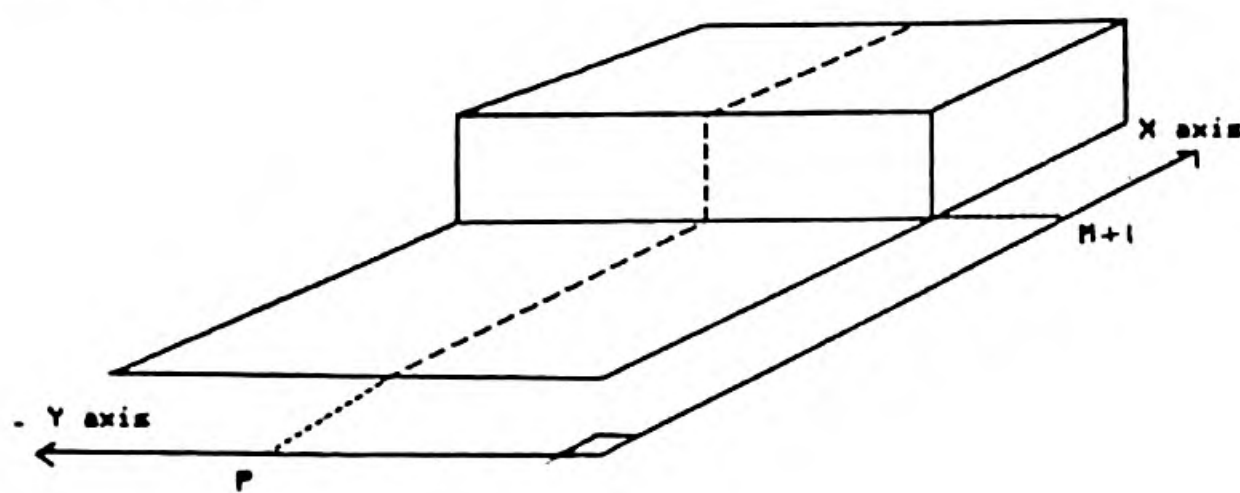


Fig. 4: Edge in a two dimensional signal.

$$g(i,j) = \begin{cases} 0 & \text{for } i \leq m \\ 2 & \text{for } i > m \end{cases} \quad (15)$$

The two dimensional noisy signal is given by

$$s(i,j) = g(i,j) + n(i,j) \quad (16)$$

where

$$P_r[n(i,j)] = \begin{cases} \alpha = 1 - \beta & \text{if } n(i,j) = 0 \\ \beta & \text{if } n(i,j) = 1 \end{cases} \quad (17)$$

Let the filtered sequence be $y(i,j)$ and the noise be distributed as in (6). As an example consider the case where an edge jitter equal to 1. If no edge jitter occurs at the Y-axis position p , the output $y(m,p)$ must be equal to 0. If an impulse occurs at position m , then $y(m,p) = 1$ and $y(m-1,p) = 0$. A rising edge is a root signal meaning that the separable median filter will not be able to eliminate this impulse. It follows that the original edge is shifted on the left by one position. Larger amounts of edge jitter can occur as well. We must also take into account in the edge jitter analysis that there are two ways of separable filtering the noisy signal: we can filter the X-axis first and then the Y-axis, also, we can filter the Y-axis first and the X-axis second. The edge jitter introduced by these two different cases will be in general different.

CASE I: X-Y Filtering

We now consider the X-Y separable recursive median filter. Before the window reaches the edge, the recursive median filtering is a stationary process; its output distribution is given by (7) and (8). Once the window of the filter contains the edge, the process becomes nonstationary and can only be described by point to point transition matrices. For the n nonstationary positions before the rising edge, the generalized form of these matrices is ⁷:

$$\begin{matrix} & 0_{m-k} & 1_{m-k} \\ 0_{m-k-1} & \begin{bmatrix} 1 - \beta^{k+1} & \beta^{k+1} \\ 0 & 1 \end{bmatrix} \\ 1_{m-k-1} & \end{matrix} \quad (18)$$

for $0 \leq k \leq n-1$

Assume that the sample to be filtered is at position $(m-k)$ with $0 \leq k \leq n-1$, and let

$$\pi_{m-k} = [\pi_{m-k}(0), \pi_{m-k}(1)] \quad (19)$$

be the vector defining the output distribution of the recursive median filter at the nonstationary position $(m-k)$. To compute π_{m-k} we can use the relation ⁶:

$$\pi_{m-k} = \pi_{1d} P \quad (20)$$

where

$$\pi_{1d} = [\pi_{1d}(0), \pi_{1d}(1)] \quad (21)$$

is the vector formed with equations (7) and (8), and where the matrix P

$$P = \begin{bmatrix} \prod_{i=0}^t (1 - \beta^{i+1}) & \sum_{r=0}^t \left\{ \left[\prod_{i=0}^r (1 - \beta^i) \right] \beta^{r+1} \right\} \\ 0 & 1 \end{bmatrix} \quad (22)$$

with $t = n-1-k$

is the matrix resulting from the product of the $(n-k+1)$ matrices given in (18) that describe the transition probabilities from positions $m-(n-1)$ to $m-k$. Hence,

$$\pi_{m-k}(0) = \pi_{1d}(0) \left\{ \prod_{i=0}^t (1 - \beta^{i+1}) \right\} \quad (23a)$$

$$\pi_{m-k}(1) = \pi_{1d}(0) \left\{ \sum_{r=0}^t \left\{ \left[\prod_{i=0}^r (1 - \beta^i) \right] \beta^{r+1} \right\} \right\} + \pi_{1d}(1) \quad (23b)$$

with $t = n-1-k$ and for $0 \leq k \leq n-1$.

Equations (23a) and (23b) define the vector (19). At position $(m+1)$ the transition probability matrix takes the following particular form:

$$\begin{matrix} & 2_{m+1} & 3_{m+1} \\ \begin{matrix} 0_m \\ 1_m \end{matrix} & \begin{bmatrix} 1-\beta^{n+1} & \beta^{n+1} \\ 1-\beta^{n+1} & \beta^{n+1} \end{bmatrix} \end{matrix} \quad (24)$$

Again we compute π_{m+1} to determine the distribution at the output of the recursive median filter:

$$\pi_{m+1} = \pi_{1d} P, \quad (25)$$

where P is now the product of all the matrices (18) and (24),

$$P = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad (26)$$

where,

$$A = \prod_{i=0}^n (1-\beta^{i+1}) + (1-\beta^{n+1}) \sum_{r=0}^{n-1} \left\{ \prod_{i=0}^r (1-\beta^i) \right\} \beta^{r+1}, \quad (27a)$$

$$B = \beta^{n+1} \left[\prod_{i=0}^{n-1} (1-\beta^{i+1}) + \sum_{r=0}^{n-1} \left\{ \prod_{i=0}^r (1-\beta^i) \right\} \beta^{r+1} \right], \quad (27b)$$

$$C = 1 - D = 1 - \beta^{n+1}, \quad (27c)$$

Using (25), π_{m+1} is the vector formed with the following terms:

$$\pi_{m+1}(2) = A\pi_{1d}(0) + C\pi_{1d}(1), \quad (28a)$$

$$\pi_{m+1}(3) = B\pi_{1d}(0) + D\pi_{1d}(1) \quad (28b)$$

This vector defines the output distribution of the X-axis recursively median filtered signal at position $(m+1)$. The $(n-1)$ remaining nonstationary positions are found following the rising edge. The generalized form of the matrices in that area is shown below.

$$\begin{matrix} & 2_{m+1+k} & 3_{m+1+k} \\ \begin{matrix} 0_{m+k} \\ 1_{m+k} \end{matrix} & \begin{bmatrix} 1-\beta^{n+1} & \beta^{n+1} \\ 0 & 1 \end{bmatrix} \end{matrix} \quad (29)$$

with $1 \leq k \leq n-1$ and $n > 1$

The vector π_{m+1+k} defining the output distribution at the position $(m+1+k)$ is found using

$$\pi_{m+1+k} = \pi_{1d} P, \quad (30)$$

where

$$P = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \times \begin{bmatrix} (1-\beta^{n+1})^k & \beta^{n+1} \sum_{i=1}^k [(1-\beta^{n+1})^{i-1}] \\ 0 & 1 \end{bmatrix} \quad (31)$$

Hence,

$$P = \begin{bmatrix} AC^k & AD \sum_{i=1}^k [C^{i-1}] + B \\ C^{k+1} & D \left[C \sum_{i=1}^k [C^{i-1}] + 1 \right] \end{bmatrix} \quad (32)$$

For simplicity, let

$$P = \begin{bmatrix} E & F \\ G & H \end{bmatrix} \quad (33)$$

The vector (30) defining the output distribution at position $(m+1+k)$, with $0 \leq k \leq n-1$ and $n > 1$, is formed with the following terms:

$$\pi_{m+k+1}(2) = E\pi_{1d}(0) + G\pi_{1d}(1), \quad (34a)$$

$$\pi_{m+k+1}(3) = F\pi_{1d}(0) + H\pi_{1d}(1). \quad (34b)$$

After the window of the median filter is slid across the entire nonstationary region, a new stationary region is encountered. The distribution at the output of the filter is simply given by (7) and (8).

Once the filtering along the X-axis is done, the filtering along the Y-axis is next. Let $\pi_x(\text{sgn}(x-j)+1)$ and $\pi_x(\text{sgn}(x-j)+2)$ be the probability of a sample along the x coordinate, where $\text{sgn}(t)=1$ if $t>0$ and $\text{sgn}(t)=-1$ if $t<0$, then referring to (7) and (8) the output distribution for the output of the separable filter is

$$\pi_{2d}(\text{sgn}(x-m-1)+1) = \quad (35a)$$

$$\frac{\pi_x^{n+1}(\text{sgn}(x-m-1)+1) + n\pi_x^{n+1}(\text{sgn}(x-m-1)+1)\pi_x^{n+1}(\text{sgn}(x-m-1)+2)}{\pi_x^{n+1}(\text{sgn}(x-m-1)+1) + 2n\pi_x^{n+1}(\text{sgn}(x-m-1)+1)\pi_x^{n+1}(\text{sgn}(x-m-1)+2) + \pi_x^{n+1}(\text{sgn}(x-m-1))}$$

$$\pi_{2d}(\text{sgn}(x-m-1)+2) = \quad (35b)$$

$$\frac{\pi_x^{n+1}(\text{sgn}(x-m-1)+2) + n\pi_x^{n+1}(\text{sgn}(x-m-1)+1)\pi_x^{n+1}(\text{sgn}(x-m-1)+2)}{\pi_x^{n+1}(\text{sgn}(x-m-1)+1) + 2n\pi_x^{n+1}(\text{sgn}(x-m-1)+1)\pi_x^{n+1}(\text{sgn}(x-m-1)+2) + \pi_x^{n+1}(\text{sgn}(x-m-1))}$$

for samples at any Y coordinate, where π_x is given by either the expression in (7), (8), (23a), (23b), (28a), (28b), (34a) or (34b), depending on the value of x.

A similar analysis is done to compute the falling edge jitter statistics. We will only mention below the matrices that have been used to describe the separable median filtering in the nonstationary region about the falling edge.

$$\begin{matrix} & 2_{m-k} & 3_{m-k} \\ \begin{matrix} 2_{m-k-1} \\ 3_{m-k-1} \end{matrix} & \begin{bmatrix} 1 & 0 \\ (1-\beta)^k & 1-(1-\beta)^k \end{bmatrix} \end{matrix}$$

for $1 \leq k \leq n$

$$\begin{matrix} & 0_m & 1_m \\ \begin{matrix} 2_{m-1} \\ 3_{m-1} \end{matrix} & \begin{bmatrix} (1-\beta)^{n+1} & 1-(1-\beta)^{n+1} \\ (1-\beta)^{n+1} & 1-(1-\beta)^{n+1} \end{bmatrix} \end{matrix}$$

$$\begin{matrix} & 0_{m+k} & 1_{m+k} \\ \begin{matrix} 0_{m+k-1} \\ 1_{m+k-1} \end{matrix} & \begin{bmatrix} 1 & 0 \\ (1-\beta)^{n+1} & 1-(1-\beta)^{n+1} \end{bmatrix} \end{matrix}$$

for $1 \leq k \leq n$

With the help of all the expressions above, we are able to compute the probability of the edge jitter. To compare the filtering performance of the standard filter versus that of the recursive, we use the same four level model as chosen previously. After computing the statistics for the edge jitter for the standard median filter, the variance of the edge jitter is obtained for different window lengths. Table 1 shows the variance of the edge jitter for a window of size 3. Unlike the standard filter, the variance of the rising edge and falling edge are not the same when the recursive median filter is used.

β	$n = 1$		
	Std	Rec R	Rec F
0.1e-05	9.0e-20	6.0e-20	3.2e-19
0.1e-04	9.0e-16	6.0e-16	3.2e-15
0.1e-03	9.0e-12	6.0e-12	3.2e-11
0.2e-03	1.4e-10	1.0e-10	5.3e-10
0.5e-03	5.6e-09	4.2e-09	2.2e-08
0.1e-02	8.9e-08	7.5e-08	3.8e-07
0.2e-02	1.4e-06	1.5e-06	7.2e-06
0.5e-02	5.1e-05	9.9e-05	4.0e-04
0.1e-01	7.0e-04	3.3e-03	9.7e-03
0.2e-01	6.9e-03	1.5e-01	2.4e-01

β = Probability of positive impulse.
Std = Standard median filter.
Rec = Recursive median filter - R = Rising edge - F = Falling edge
 $2n + 1$ = window length

TABLE 1
FOR THE SEPARABLE MEDIAN FILTER X-Y:
VARIANCE OF THE RISING AND FALLING EDGES VERSUS
VERSUS THE PROBABILITY OF POSITIVE IMPULSES

CASE II: Y-X Filtering

We now consider the Y-X separable recursive median filter: we first filter the Y-axis and the X-axis second. Each column in the Y-axis is a constant plus noise; the output distribution of the signal at this first stage is simply given by equations (7) and (8). The filtering of the edge is then done on the X-axis. Referring to the previous case, we can state that there are two stationary regions respectively before and after the rising edge. The output distribution is:

$$\pi_{2d}(\text{sgn}(x-m-1)+1) = \quad (36a)$$

$$\frac{\pi_{1d}^{n+1}(\text{sgn}(x-m-1)+1) + n\pi_{1d}^{n+1}(\text{sgn}(x-m-1)+1)\pi_{1d}^{n+1}(\text{sgn}(x-m-1)+2)}{\pi_{1d}^{n+1}(\text{sgn}(x-m-1)+1) + 2n\pi_{1d}^{n+1}(\text{sgn}(x-m-1)+1)\pi_{1d}^{n+1}(\text{sgn}(x-m-1)+2) + \pi_{1d}^{n+1}(\text{sgn}(x-m-1)+2)}$$

$$\pi_{2d}(\text{sgn}(x-m-1)+2) = \quad (36b)$$

$$\frac{\pi_{1d}^{n+1}(\text{sgn}(x-m-1)+2) + n\pi_{1d}^{n+1}(\text{sgn}(x-m-1)+1)\pi_{1d}^{n+1}(\text{sgn}(x-m-1)+2)}{\pi_{1d}^{n+1}(\text{sgn}(x-m-1)+1) + 2n\pi_{1d}^{n+1}(\text{sgn}(x-m-1)+1)\pi_{1d}^{n+1}(\text{sgn}(x-m-1)+2) + \pi_{1d}^{n+1}(\text{sgn}(x-m-1)+2)}$$

for any $x \leq (m-n)$ or $x > (m+n)$ and where the vector π_{1d} is defined by (7) and (8). We can also state that these two stationary regions are separated by $2n$ nonstationary positions. n of them are found from position $(m-n+1)$ to m before the rising edge and n others occupy the positions $(m+1)$ to $(m+n)$ after the rising edge. Thus, the expressions of the matrices (18),(24),(29) can be used to determine the output distribution at each of these nonstationary position. Their updated expressions for a two dimensional case are shown below. For the first n nonstationary positions before the rising edge, the transition matrices have the following form:

$$\begin{matrix} 0_{m-k} & 1_{m-k} \\ 0_{m-k-1} & 1_{m-k-1} \end{matrix} \begin{bmatrix} 1-\pi_{1d}^{k+1}(1) & \pi_{1d}^{k+1}(1) \\ 0 & 1 \end{bmatrix} \quad (38)$$

for $0 \leq k \leq (n-1)$.

The transition matrix at position $(m+1)$ takes the following particular expression:

$$\begin{matrix} 2_{m+1} & 3_{m+1} \\ 0_m & 1_m \end{matrix} \begin{bmatrix} 1-\pi_{1d}^{n+1}(1) & \pi_{1d}^{n+1}(1) \\ 1-\pi_{1d}^{n+1}(1) & \pi_{1d}^{n+1}(1) \end{bmatrix} \quad (39)$$

The $(n-1)$ last nonstationary positions are found after the rising edge. The $(n-1)$ transition matrices have the following generalized form:

$$\begin{matrix} 2_{m+1+k} & 3_{m+1+k} \\ 2_{m+k} & 3_{m+k} \end{matrix} \begin{bmatrix} 1-\pi_{1d}^{n+1}(1) & \pi_{1d}^{n+1}(1) \\ 0 & 1 \end{bmatrix} \quad (40)$$

for $1 \leq k \leq (n-1)$, and $n > 1$

Let the value of the edge jitter be i . The existence of a nonstationary region before the rising edge leads us to consider different ranges of i in the computation of the edge jitter statistics. A first range contains n values from $i=1$ to $i=n$; an edge will be shifted to the left by i positions when the $(n+1)$ root samples from position $(m-n)$ to $(m-1)$ are zero valued whereas the root samples from position $(m-1+1)$ to m are one valued. The probability of such a sequence is:

$$P_r(i=l) = \pi_{2d}(0) P_r[0_{m+1-n}/0_{m-n}] \cdots \times P_r[1_{m+1-l}/0_{m-l}] \cdots P_r[1_m/1_{m-1}] \quad (41)$$

and with the help of the matrix (38),

$$P_r(i=l) = \pi_{2d}(0) \left\{ \prod_{k=1}^{k=n-l} (1-\pi_{1d}^{k+1}(1)) \right\} \pi_{1d}^l(1) \quad (42)$$

with $1 \leq l \leq n$.

An edge jitter $i=n+1$ occurs when the last position of the stationary region (position $(m-n)$) as well as the n following nonstationary positions are of value 1. Moreover, the median value at position $(m-(n+1))$ must be equal to zero. This implies that positions $(m-(2n+1))$ to $(m-(n+1))$, which are part of the root signal, are all zero valued. Hence, the probability that the the edge jitter i is equal to $(n+1)$ is:

$$P_r(i=n+1) = P_r[0_{m-(2n+1)}, \dots, 0_{m-(n+1)}] \times P_r[0_{m-2n}, \dots, 0_{m-(n+1)}, 1_{m-n}/0_{m-(2n+1)}, \dots, 0_{m-(n+1)}] \times P_r[1_{m-(n+1)}/1_{m-n}] \times \cdots \times P_r[1_m/1_{m-1}] \quad (43)$$

At this point we notice that we can use state transition probabilities for the positions from the stationary region: a sequence of n filtered zero values defines a "A" state, and a sequence of $(n-1)$ zeroes followed by a one valued sample defines a "B(1)" state^{4,5}. Therefore,

$$P_r(i=n+1) = \pi_{1d}(A) P_r[B(1)_{m-n}/A_{m-(n+1)}] \times P_r[1_{m-(n+1)}/1_{m-n}] \times \cdots \times P_r[1_m/1_{m-1}] \quad (44)$$

Finally, we can use again the matrix given by (38)

$$P_r(i=n+1) = \frac{\pi_{1d}^{n+1}(0)}{\pi_{1d}^{n+1}(0) + 2n\pi_{1d}^{n+1}(0)\pi_{1d}^{n+1}(1) + \pi_{1d}^{n+1}(1)} \pi_{1d}^{n+1}(1) \quad (45)$$

It can be shown that the probability of an edge jitter i equal to l with $n+2 \leq l \leq 2n+1$ has a similar expression. This is due to the fact that the "B" states $B(1)$ to $B(l-n)$ are encountered for an edge jitter l of value between $n+2$ and $2n$; the case $l=2n+1$ includes in addition a transition from the state "D" to the state "B(n)". All these state transition probabilities have been already found to be equal to one⁷. For values of the edge jitter i greater than $l=2n+1$, the shifted edge propagates in the stationary region through "D" states. Therefore the probability that the edge jitter is equal to $l \geq 2n+2$ is:

$$P_r(i=l) = \frac{\pi_{1d}^{n+1}(0)\pi_{1d}^{n+1}(1)}{\pi_{1d}^{n+1}(0) + 2n\pi_{1d}^{n+1}(0)\pi_{1d}^{n+1}(1) + \pi_{1d}^{n+1}(1)} \times \left\{ 1-\pi_{1d}^{n+1}(0) \right\}^{l-(2n+1)} \quad (46)$$

The case of the falling edge is also treated in a similar fashion. The point to point matrices for the nonstationary samples around the falling edge are shown below.

$$\begin{matrix} 2_{m-k} & 3_{m-k} \\ 2_{m-k-1} & 3_{m-k-1} \end{matrix} \begin{bmatrix} 1 & 0 \\ \pi_{1d}(0)^k & 1-\pi_{1d}(0)^k \end{bmatrix}$$

for $1 \leq k \leq n$

$$\begin{matrix} 0_m & 1_m \\ 2_{m-1} & 3_{m-1} \end{matrix} \begin{bmatrix} \pi_{1d}(0)^{n+1} & 1-\pi_{1d}(0)^{n+1} \\ \pi_{1d}(0)^{n+1} & 1-\pi_{1d}(0)^{n+1} \end{bmatrix}$$

$$\begin{matrix} 0_{m+k} & 1_{m+k} \\ 0_{m+k-1} & 1_{m+k-1} \end{matrix} \begin{bmatrix} 1 & 0 \\ \pi_{1d}(0)^{n+1} & 1-\pi_{1d}(0)^{n+1} \end{bmatrix}$$

for $1 \leq k \leq n$

The variance of the edge jitter has been computed for selected values of β ; the results are shown in table 2. Note that the rising edge jitter obtained for the recursive filter is smaller than for the standard filter for both window sizes, meaning that the recursive median filter preserves rising edges better than the standard median filter. Conversely the variance of the falling edge jitter for the recursive median filter is slightly greater than for the standard median filter. Also, we can notice that the original symmetry between a rising and a falling edge is not preserved by the recursive median filtering process.

β	n = 1			n=2		
	Std	Rec R	Rec F	Std	Rec R	Rec F
0.1e-05	8.0e-10	2.0e-10	4.0e-10	3.0e-14	3.0e-15	9.0e-15
0.1e-04	8.0e-08	2.0e-08	4.0e-08	3.0e-11	3.0e-12	9.0e-12
0.1e-03	8.0e-06	2.0e-06	4.0e-06	3.0e-08	3.0e-09	9.0e-09
0.2e-03	2.4e-05	8.0e-06	1.6e-05	2.4e-07	2.4e-08	7.2e-08
0.5e-03	1.5e-04	5.0e-05	1.0e-04	3.7e-08	3.8e-07	1.1e-06
0.1e-02	8.0e-04	2.0e-04	4.0e-04	3.0e-05	3.0e-06	9.1e-06
0.2e-02	2.4e-03	8.2e-04	1.6e-03	2.3e-04	2.5e-05	7.3e-05
0.5e-02	1.5e-02	5.6e-03	1.1e-02	3.5e-03	4.0e-04	1.2e-03
0.1e-01	8.3e-02	2.7e-02	4.7e-02	2.6e-02	3.4e-03	1.0e-02
0.2e-01	3.2e-01	2.0e-01	2.5e-01	2.0e-01	3.4e-02	1.1e-01

β = Probability of positive impulse.
Std : Standard median filter.
Rec : Recursive median filter - R : Rising edge - F : Falling edge.
 $2n + 1$ = window length.

TABLE 2
FOR THE SEPARABLE MEDIAN FILTER Y-X:
VARIANCE OF THE RISING AND FALLING EDGES VERSUS
THE PROBABILITY OF POSITIVE IMPULSES

2) Edge jitter due to positive and negative impulsive type noise.

Consider an original two level signal that is perturbed by either positive or negative impulses. Hence we have a six level signal with two kinds of edge jitter: a rising edge can be shifted to the left because of the occurrence of positive impulses on its left. But a rising edge can be also shifted to the right because of negative impulses occurring on its right. Using a similar analysis as before, we compute the variance of the rising and the falling edge for a window

β	n = 1			n=2		
	Std	Rec R	Rec F	Std	Rec R	Rec F
0.1e-05	1.2e-09	6.0e-10	6.0e-10	6.0e-14	1.2e-14	1.2e-14
0.1e-04	1.2e-07	6.0e-08	6.0e-08	6.0e-11	1.2e-11	1.2e-11
0.1e-03	1.2e-05	6.0e-06	6.0e-06	6.0e-08	1.2e-08	1.2e-08
0.2e-03	4.8e-05	2.4e-05	2.4e-05	4.8e-07	9.6e-08	9.6e-08
0.5e-03	3.0e-04	1.5e-04	1.5e-04	7.4e-06	1.5e-06	1.5e-06
0.1e-02	1.2e-03	6.1e-04	6.1e-04	5.9e-05	1.2e-05	1.2e-05
0.2e-02	4.7e-03	2.5e-03	2.5e-03	4.7e-04	9.9e-05	9.9e-05
0.5e-02	2.9e-02	1.6e-02	1.6e-02	6.9e-03	1.6e-03	1.6e-03
0.1e-01	1.2e-01	7.4e-02	7.5e-02	5.0e-02	1.4e-02	1.4e-02
0.2e-01	5.0e-01	4.4e-01	4.5e-02	3.2e-01	1.5e-01	1.5e-01

β = Probability of positive impulse.
Std : Standard median filter.
Rec : Recursive median filter - R : Rising edge - F : Falling edge.
 $2n + 1$ = window length.

TABLE 3
FOR THE SEPARABLE MEDIAN FILTER Y-X:
VARIANCE OF THE RISING AND FALLING EDGES VERSUS
THE PROBABILITY OF POSITIVE AND NEGATIVE IMPULSES.

length 3 and 5. Table 3 shows the results. Notice that the standard median filtering of a signal with double sided white impulsive type noise creates more edge jitter than the recursive median filter, regardless of the nature of the edge (rising or falling edge). This confirms that the recursive median filter performs better in preserving edges at their original position in a noisy signal than the standard filter. In order to illustrate better the nonsymmetry, the expected value and the standard deviation of each recursively filtered nonstationary position has been computed. The results for the X-Y and the Y-X separable recursive median filter of window length 3 are plotted respectively in Fig. 5 and Fig. 6; the probability of positive or negative impulses has been chosen to $\beta=0.01$.

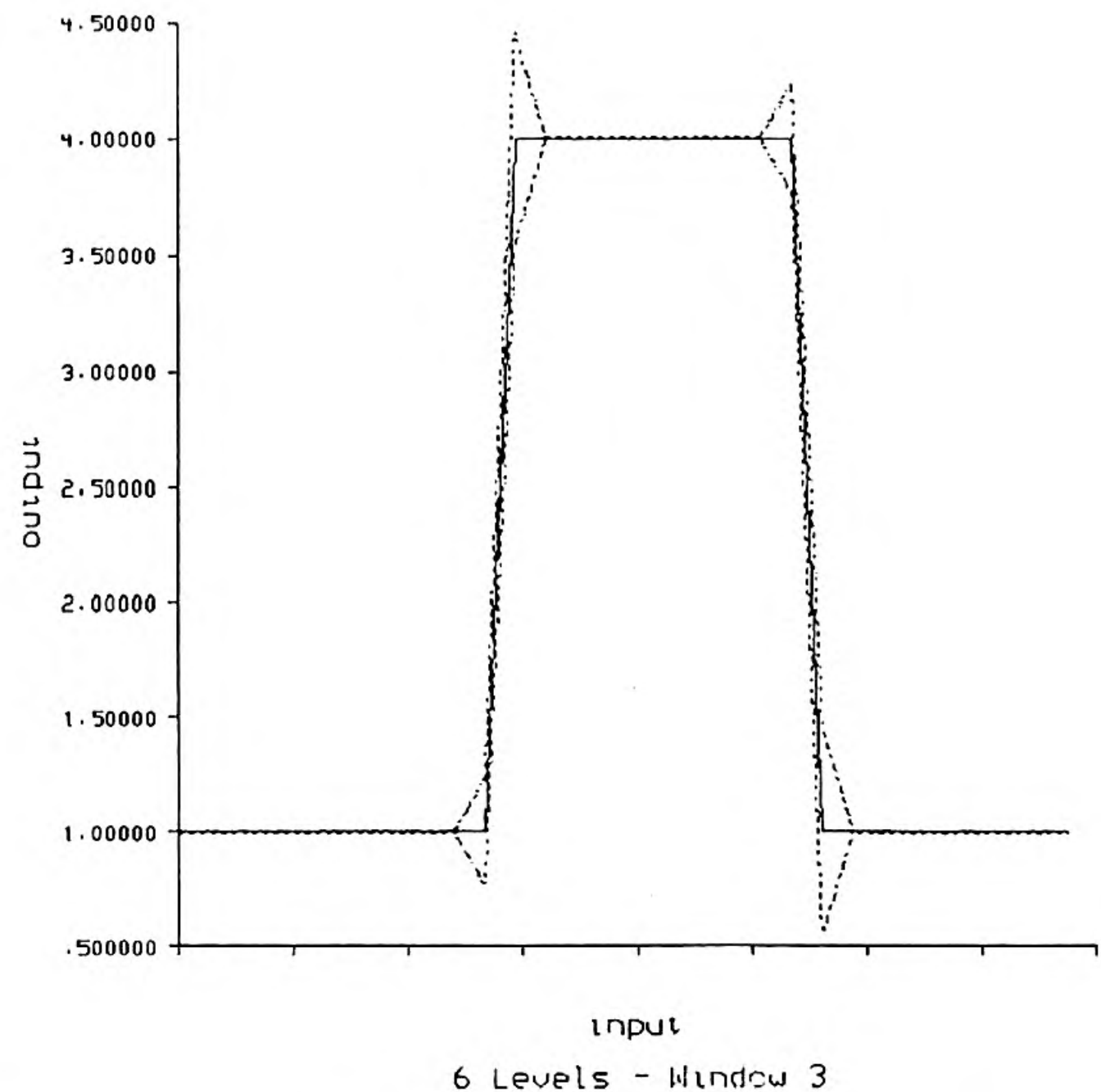


Fig. 5: For output y (line -) of a X-Y separable recursive median filter with an input signal plus positive or negative impulsive type noise of probability $\beta=0.01$, the $E[y]+16\sigma$ and the $E[y]-16\sigma$ (lines ...).

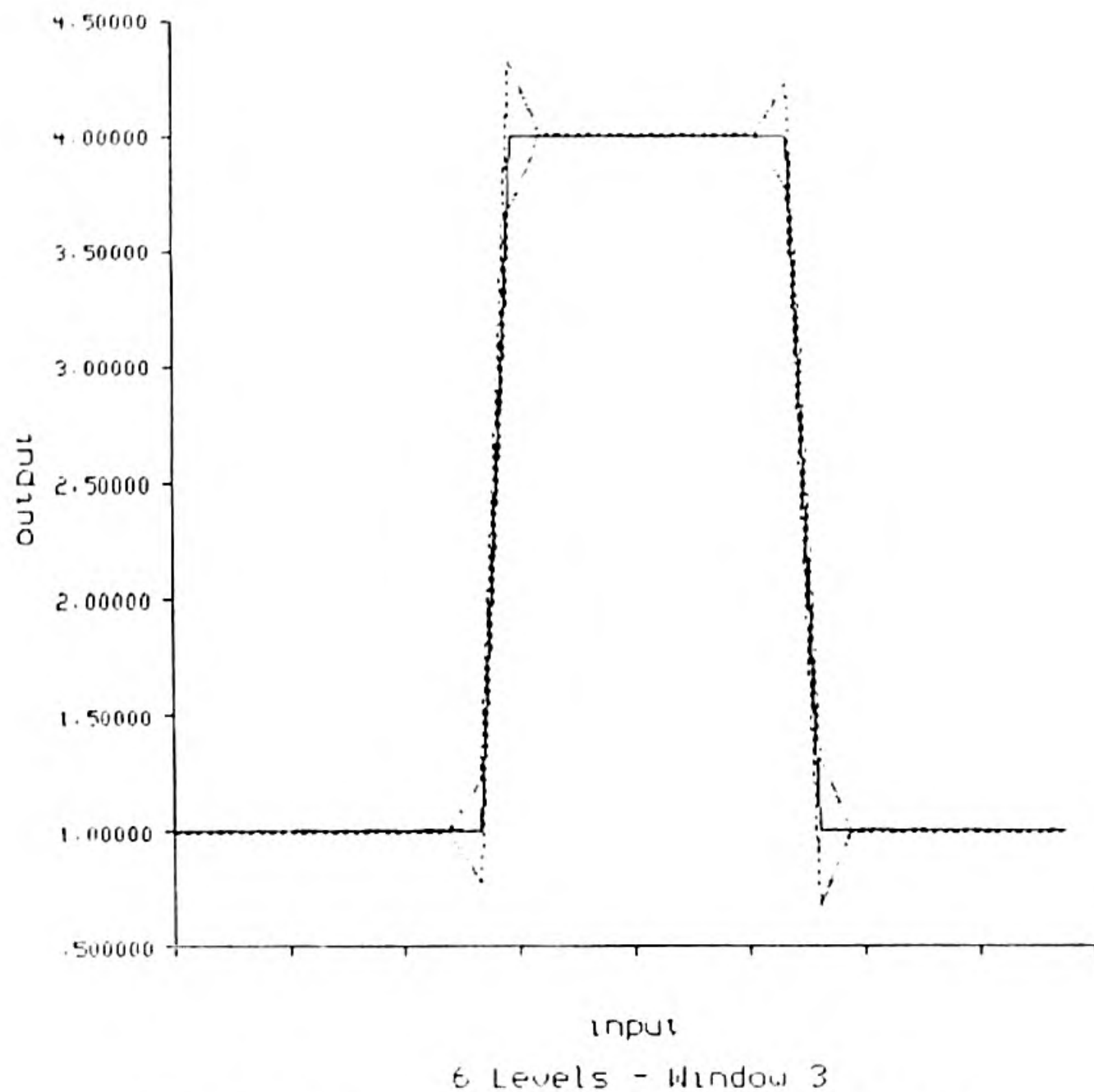


Fig. 6: For output y (line -) of a Y-X separable recursive median filter with an input signal plus positive or negative impulse type noise of probability $\beta=0.01$, the $E[y]+16\sigma$ and the $E[y]-16\sigma$ (lines ...).

IV CONCLUSION

The recursive separable median filtering of constant signals perturbed by single sided impulsive noise has been analyzed. In particular, we derived the output distribution of a recursive separable median filter of any window length in order to determine the probability of positive impulses in a filtered two dimensional signal. These results have been then compared to similar results obtained for the standard separable median filter and it is shown that the recursive median filter performs better than the standard median filter in eliminating positive impulsive type noise. The analysis has been then extended to the case of white double sided impulsive type noise and proved again that the recursive separable median filter is more efficient than the standard filter. Then a detailed analysis of the edge jitter generated from the separable filtering of a noisy edge has been presented for white single sided and double sided impulsive type noise. Again the statistical description for the recursive and standard have been compared. We showed that the recursive median preserves better the original two dimensional edge in a signal but generates a nonsymmetry between rising and falling edges.

REFERENCES

- [1] N. C. Gallagher and G. L. Wise, "A Theoretical Analysis of the Properties of Median Filters" IEEE Trans. on ASSP, vol ASSP -29, no. 6, December 1981
- [2] T. A. Nodes and N. C. Gallagher, "Median Filters: Some Modifications and their Properties" IEEE Trans. on ASSP, vol ASSP -30, no. 5, October 1982
- [3] H.A. David, "Order Statistics" Wiley, New York 1970.
- [4] G. R. Arce and N. C. Gallagher, "State Description for the Root-Signals Set of Median Filters" IEEE Trans. on ASSP, vol ASSP -30, no. 6, December 1982.

- [5] G. R. Arce and N. C. Gallagher, "Stochastic Analysis for the Multilevel Root-Signal Set of Median Filters" Proceedings of the 16th annual Conference on Information Sciences and Systems. Princeton University, March 1982
- [6] H.A. David, "Order Statistics" Wiley, New York 1970.
- [7] R.J. Crinon and G.R. Arce, "Median Filters: Analysis for Signals with additive impulsive noise" Proceedings of the 21st Allerton Conference. University of Illinois, October 1983.

A MEDIAN FILTER ARCHITECTURE SUITABLE FOR VLSI IMPLEMENTATION

*Peter J. Warter, Gary S. Delp, Jorge L. Garcia-Colevatti
John Dalton, John Baumeister*

Department of Electrical Engineering, University of Delaware
Newark, Delaware 19716

Abstract

The Median Filter is a non linear Digital filter especially useful in extracting features from noisy images. A scheme for implementing the filter with a stored-ordered list is described and a first version in VLSI NMOS is analyzed for size and timing considerations. A comparison with an alternate systolic architecture is made. This implementation scales very well for large window sizes. Real time video processing should be possible.

Introduction

The median filter is a nonlinear digital filter especially useful in extracting features from noisy images. The input to the median filter is a sequence of sample points, usually obtained by sampling an image at equally spaced intervals along a line through an image. The output of the median filter is the median of the last n (an odd number) samples input, that is the mid ranked sample in the n sample window. The use of the median as a function that discriminates against image noise while preserving the edge definition of objects in the image was first proposed by Tukey[1]. A survey of median filter design and applications can be found in Gallagher and Arce [2].

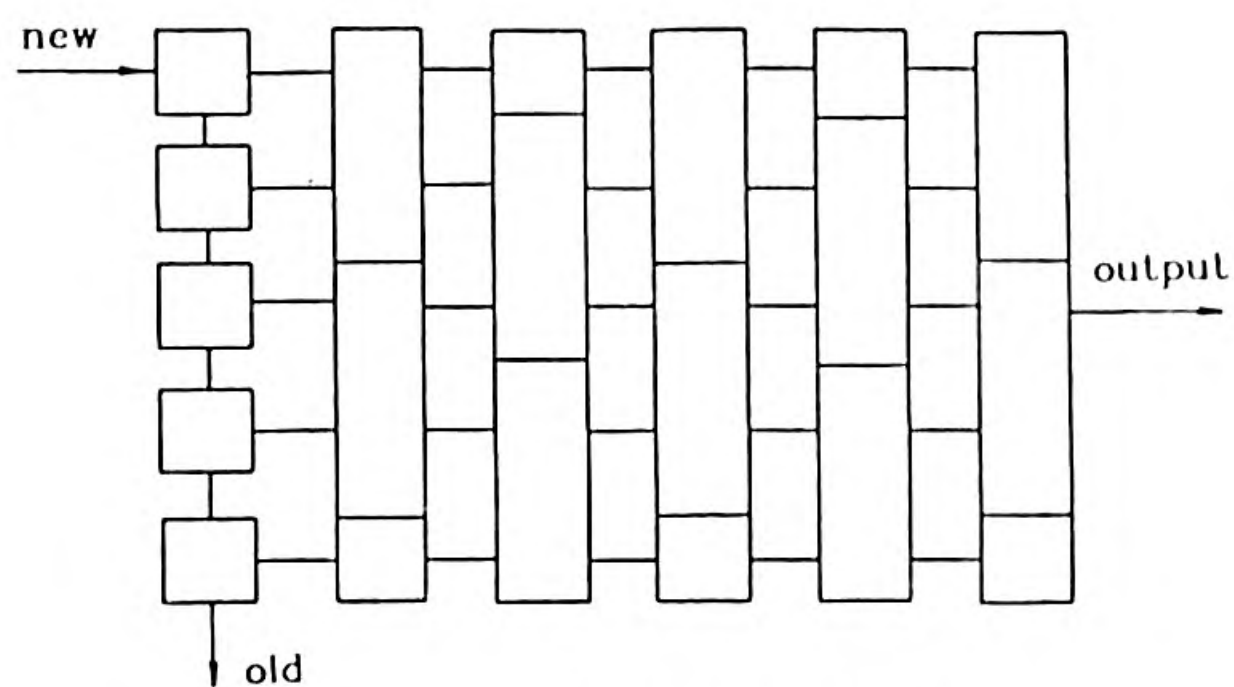
The median filter can be non-recursive, producing the median of the last n inputs; or recursive, producing the median of the last $(n+1)/2$ inputs and the last $(n-1)/2$ outputs. The recursive median filter produces a stable output in one pass, that is one that does not change with further passing of the filter over the output, and often provides better image enhancement than a single pass of a non-recursive median filter. The relative merits of recursive and non-recursive filters has been discussed by Arce and Crinon [3]. The median filter idea can be generalized to a rank order filter in which the output can be from any position in a list of the last n inputs ordered by value. Such a rank order filter might be used as a peak or valley follower.

The median filter described in this paper is one which keeps an ordered list of the last n inputs (or inputs and outputs in the case of a recursive filter). An input sequence number (modulo n) is associated with each element in the list so that the oldest can be discarded when a new sample is added to the n element list. Each sample value register in this design has a comparator so that a new sample can be compared with all of the current values at the same time. The opening of the list to allow insertion of the new value at the correct place and the discarding of the oldest value are accomplished by local control units associated with each register. This local control acts largely in response to the value comparisons made in the associated register and its neighbors. This design is relatively fast and adaptable. The window size and the rank of the element outputted can be programmed. The same filter can be programmed to be recursive or non-recursive.

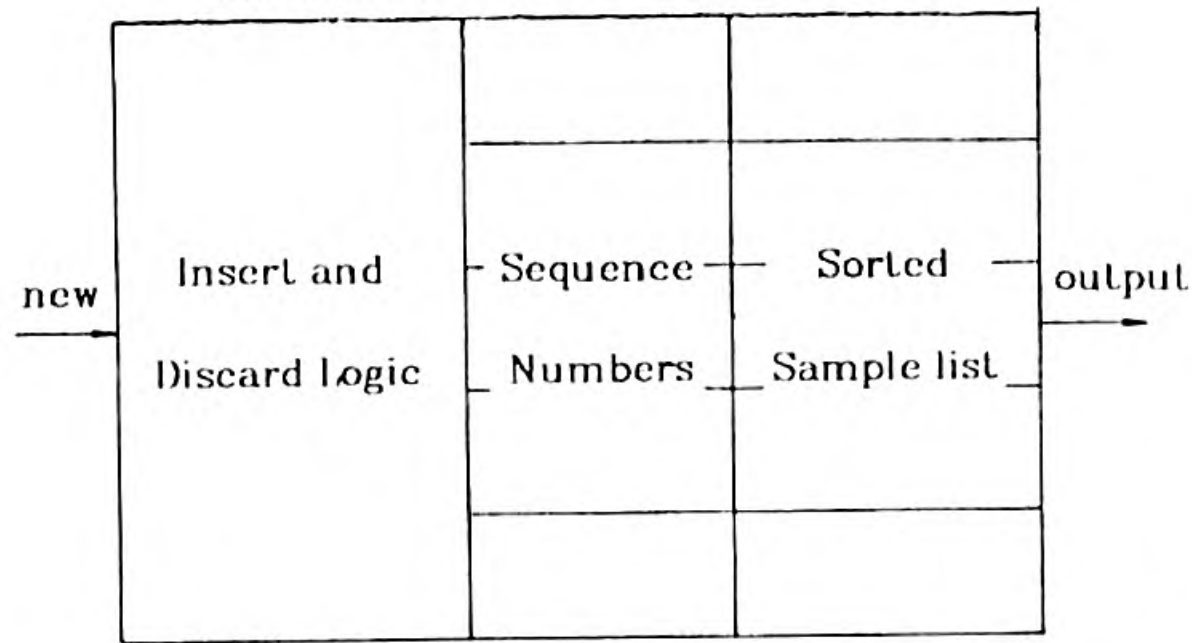
MEDIAN FILTER ARCHITECTURES

A median filter must be able to store the n samples in its window. They can be stored in order of arrival, or in value rank order. If they are stored in order of arrival, a comparison tree is required to order the samples as shown in Fig. 1a and as proposed by Ofazer [4]. If they are stored in value order, it is necessary to tag each value with its order of arrival and provide circuits for inserting a new sample into its place in the list and removing the oldest sample. Such a system is shown in Fig. 1b. We have chosen the later architecture for the reasons outlined in the introduction.

Let us consider the problem of maintaining an ordered list. We must: 1) find the proper place for the new entry, 2) discard the oldest entry and 3) move the items in the list between the place for the new entry and the discarded entry. There are three cases: 1) the new item replaces directly the item discarded, 2) the place for the new item is above the place of the discarded item in the list, and 3) the place of



a). Sequenced Data - Sort Network



b). Value ordered Data - Insert and Discard Network

Figure 1. Two Approaches - Median Filter Architecture

the new item is below the place of the discarded item. These are shown in Fig. 2. Note that the sequence number for the new sample is the same as that of the sample being discarded (modulo n). Note also that in making comparisons between the new sample and those in the value registers, we arbitrarily associate the equal condition with less than, and we arbitrarily assume that the largest value is at the "top" of the value ordered register chain and the smallest at the "bottom".

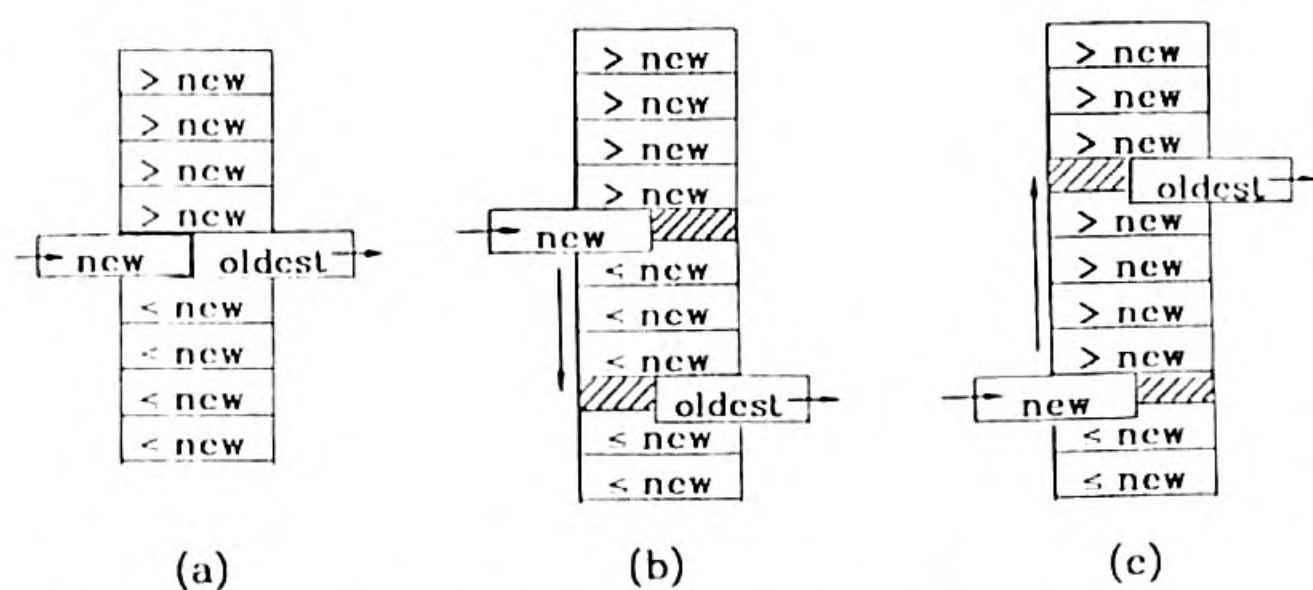


Figure 2. Possible Conditions for Updating the Ordered List

From an inspection of Fig. 2, it is obvious that on updating there are four things that can happen to each of the sample-sequence register pairs: 1) they can retain their previous values, 2) they can take on the values previously in the register pair above, 3) they can take on the values previously in the register pair below, or

4) the value register can be loaded with the new sample and the sequence register with its sequence number (modulo n).

A register retains its previous value if it is above the point in the register chain where the value comparison changes and above the sample being discarded, or if it is below the change in value comparison and below the sample being discarded. A register takes on the value of the register above in the chain if the value in the register above is less than or equal to the value of the new sample, and the register is at or above that containing the sample being discarded. A register takes on the value of the register below in the chain if the value in the register below is greater than the value of the new sample, and the register is at or below that containing the sample being discarded. A register takes on the new sample if it is at or above the sample being discarded and its value is less than or equal to the new sample and the value in the register above is greater than the that of the new sample, or if it is at or below the sample being discarded and its value is greater than the new sample and the value of the sample in the register below is less than or equal to that of the new sample.

A local control unit associated with each register determines the source of the next value-sequence number pair. This local control needs as input: 1) the value comparisons from its associated register, the one above and the one below, and 2) signals indicating whether this register is at, above or below the sample being discarded. The only non-local signal required is that indicating whether the register is above or below that containing the sample being discarded (assuming that it doesn't contain the sample to be discarded). In our system this comes from a "carry chain" that propagates a signal of one sign above the sample being discarded and the other sign below the sample to be discarded.

The filter operates sequentially. The new sample value and its sequence number (modulo n) are broadcast to the two sets of register-comparators (one for sample values, one for sequence numbers) and the comparison of the new values and those in each register pair are made locally. When the comparisons have been made and the signal has propagated on the "above/below" chain, the registers take on their new values. Then, the value register corresponding to the desired rank is the output. At this time in the sequence the filter can be made recursive by, replacing the value having a sequence number corresponding to the present output sequence number with the

output sample value. This sequence number can be generated by a modulo n counter initialized with $n / 2$.

The size of the sample window is determined by the modulo of the sequence number counter and can be made programmable, within the limits imposed by the maximum number of registers available. The rank order output can also be made programmable. There are many ways to start the filter, the simplest being to load all value registers with zero or some other standard value and to load the set of sequence registers with all allowed sequence numbers. In our design, the value registers, sequence registers, associated comparators, and local control are all the same except for the provision for automatically loading a different number in each sequence register on startup.

IMPLEMENTATION

First we will look at the control flow of the algorithm described above. Then we will look at several of the parts in detail. The control flow will be examined for critical path information. Finally, we will look at timing and scaling issues, including a comparison with an alternate architecture.

Fig. 3., which appears in the considerations sections, shows the general layout of the chip with the major functional components labeled. These include: the data, clock, and control Input Pads, the Bus Drivers, the Value and Sequence Busses (which are metal busses running vertically through the Register Set), the Value Storage Registers, the Local Control, the Sequence Number Generator, the Sequence Registers, the Vertical Carry Chain, the Rank Output Selection, Sense Amplifiers, and the Output Buffers.

The Control Flow

After the chip is setup, its function is to read new input data, update the internal list, and output the selected rank register. The data is input in a parallel fashion through standard input pads, and broadcast to the Value Bus by drivers which provide both the value and its complement to all of the registers.

At each bit of each register, the data on the bus is compared with the data in that bit of the register. The storage cell is a two inverter flip-flop circuit connected by pass transistors. These are normally connected together to form a static memory unit. During the update cycle, the inverters are separated. The input inverter reads in the new value from above, below, the bus, or its output inverter, while the output inverter keeps its old value stored in its

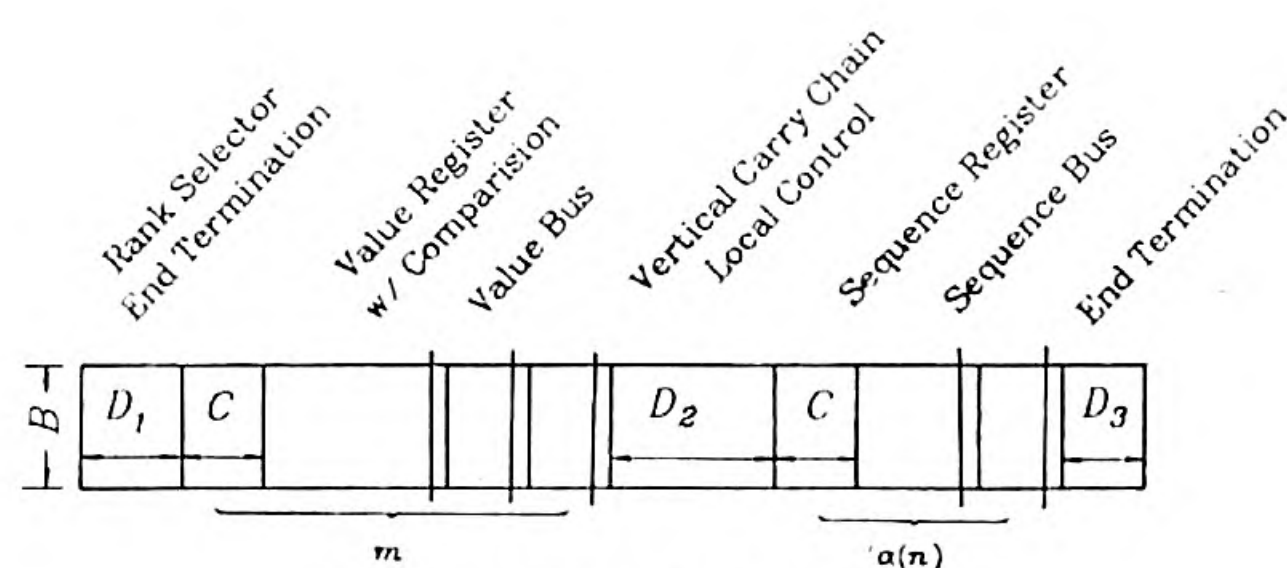
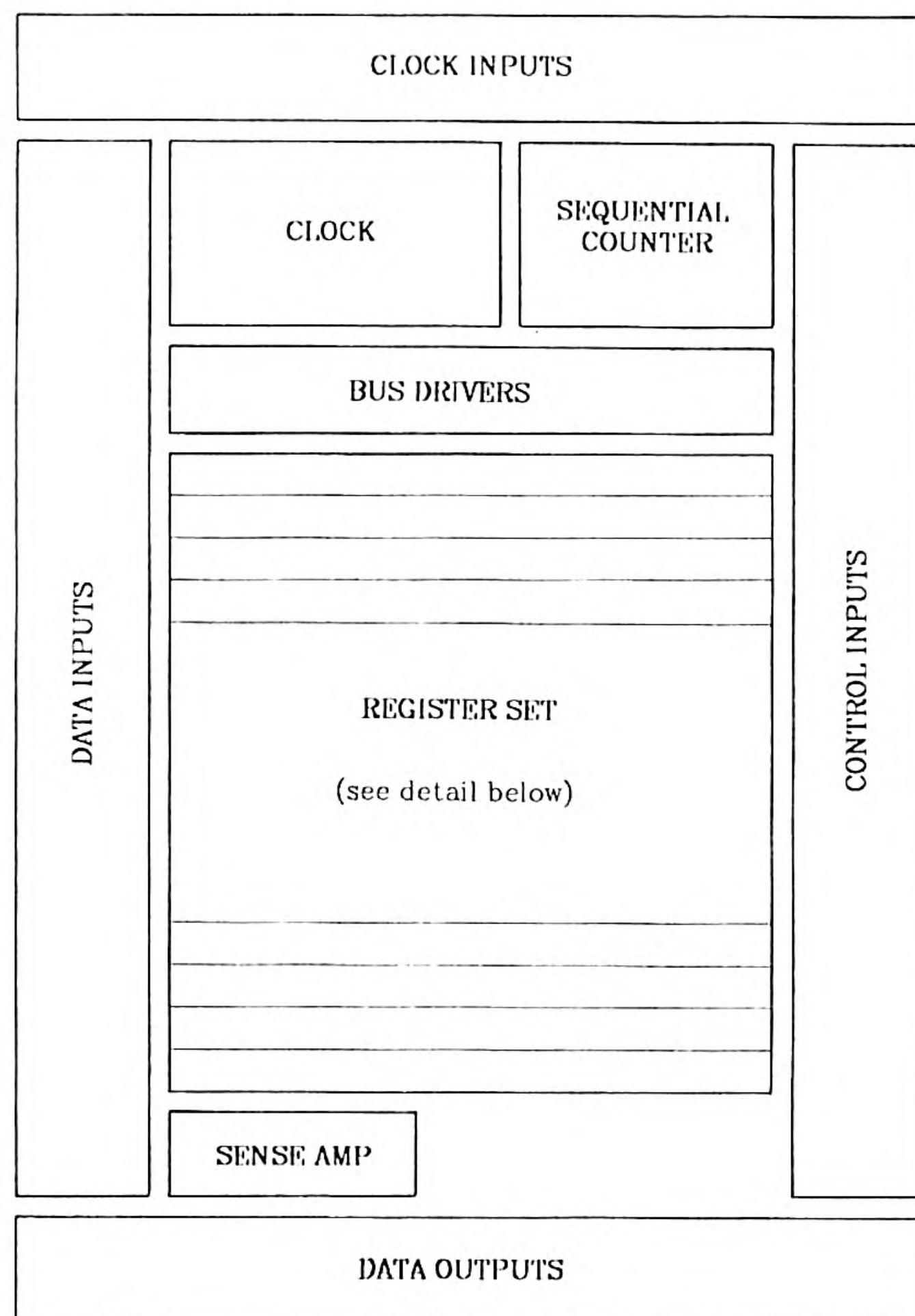


Figure 3. Floor Plan of the Chip

distributed input capacitance for the registers above or below to read. Because of this two inverter arrangement, the value in the cell is available along with its complement for the comparison circuit.

Fig. 4a shows the inverters connected as the memory cell, Fig. 4b shows the top inverter reading from its appropriate input (determined by the Local Control), and the bottom inverter providing the output for the registers above and below.

The comparison in the Value Register is an m -Bit comparison, where m is the data width. The output of the comparison is produced with

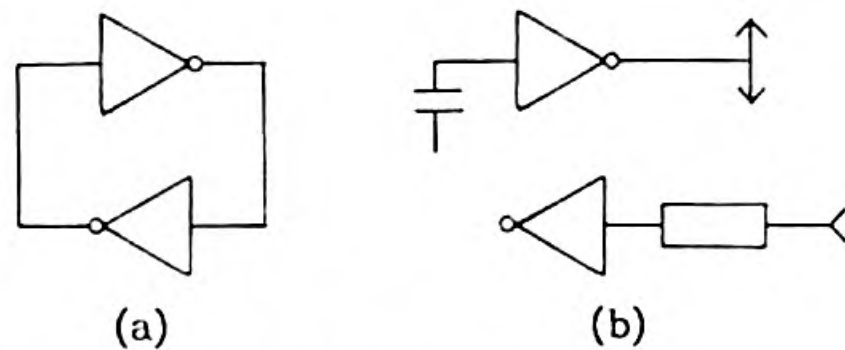
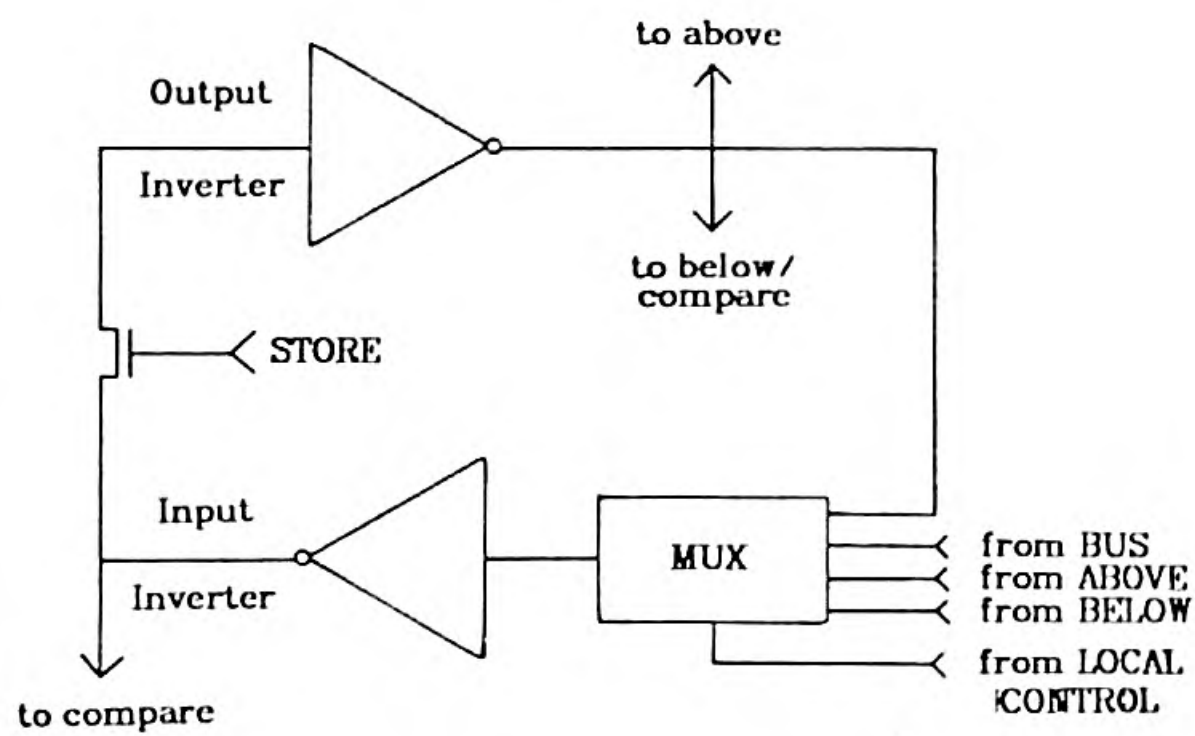


Figure 4. Storage Cell Updating

a "carry chain", which is precharged to a high level. The comparison travels from least to most significant bit. *Register less-than-or-equal-to Bus* is ground, while *Register greater-than Bus* is active high. The bitwise comparison has three possible outcomes, Register greater-than, Register equal, and Register less than the Bus. If the bits on the Bus and Register are equal, the carry chain will propagate whatever signal comes from the less significant bit. If not equal, the carry chain is isolated at this cell from the less significant cells, and the output originates from this bit. If the Register bit is less than the Bus bit, the carry chain is grounded. If the Register bit is greater than the Bus bit, the carry chain is left to float high. The input to the least significant bit is grounded when the precharge is completed and the Bus is valid, so that is all the bits are equal, the charge will be drained from the carry chain, and the comparison will yield the *less-than-or-equal-to* output. The circuit uses an exclusive-or to control the propagation of the carry chain. Because the Bus and the Register have their values and complements available at this part of the chip, the exclusive-or of those two bits is very easy to obtain. This makes this section of the circuit very simple, and the entire carry chain control includes just one powered device in each bit cell. Fig. 5 shows one cell of this comparison circuit.

The sequence number of the register to be discarded is generated by a modulo counter stepping through the allowed sequence values. This number is broadcast to the sequence registers over the Sequence Bus. A simple equivalence circuit, similar to the exclusive-or circuit in the Value Register, is used to find the

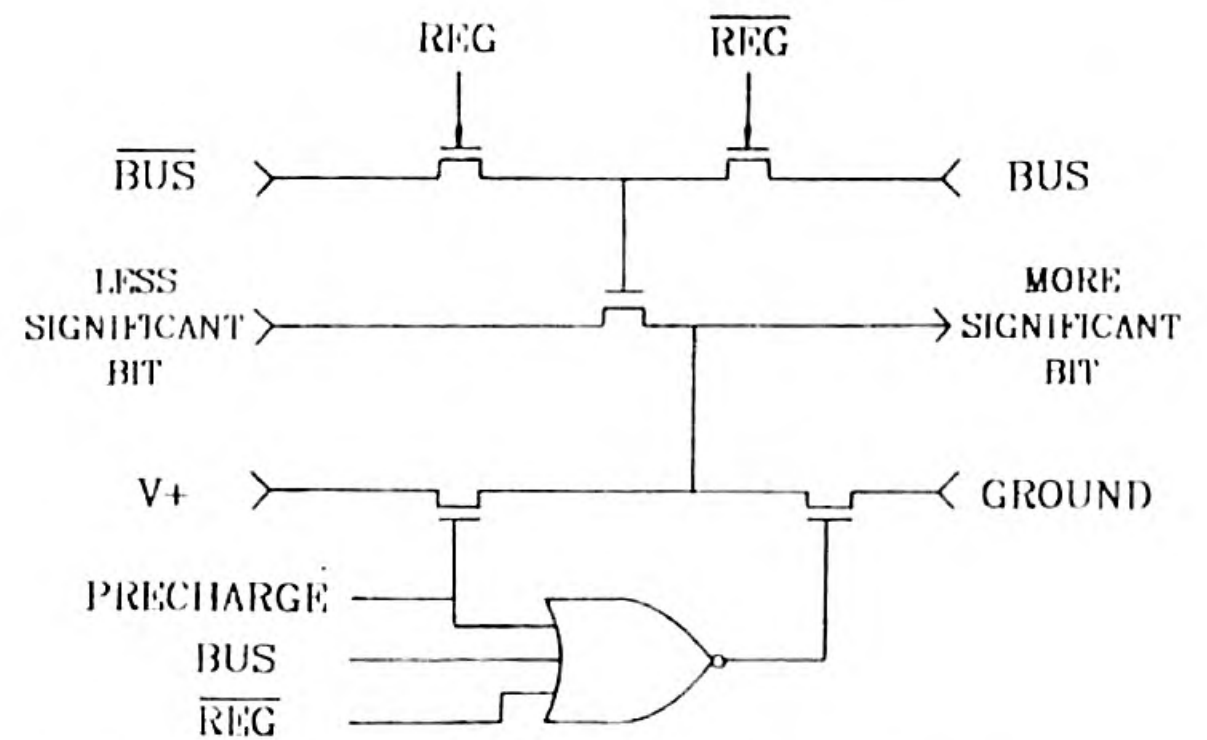


Figure 5. Value Comparison Carry Chain

matching register set. Unlike the value comparison, this information must propagate the entire height of the register set. A preset, active circuit, called the vertical carry chain, is used to speed this communication. Fig. 6 shows one register's portion of the vertical carry chain and one bit's worth of equivalence circuit. Fortunately, this comparison and propagation can begin immediately after the updating of the registers, during the Data Output cycle, so this non-local communication requirement does not limit the overall speed of the chip.

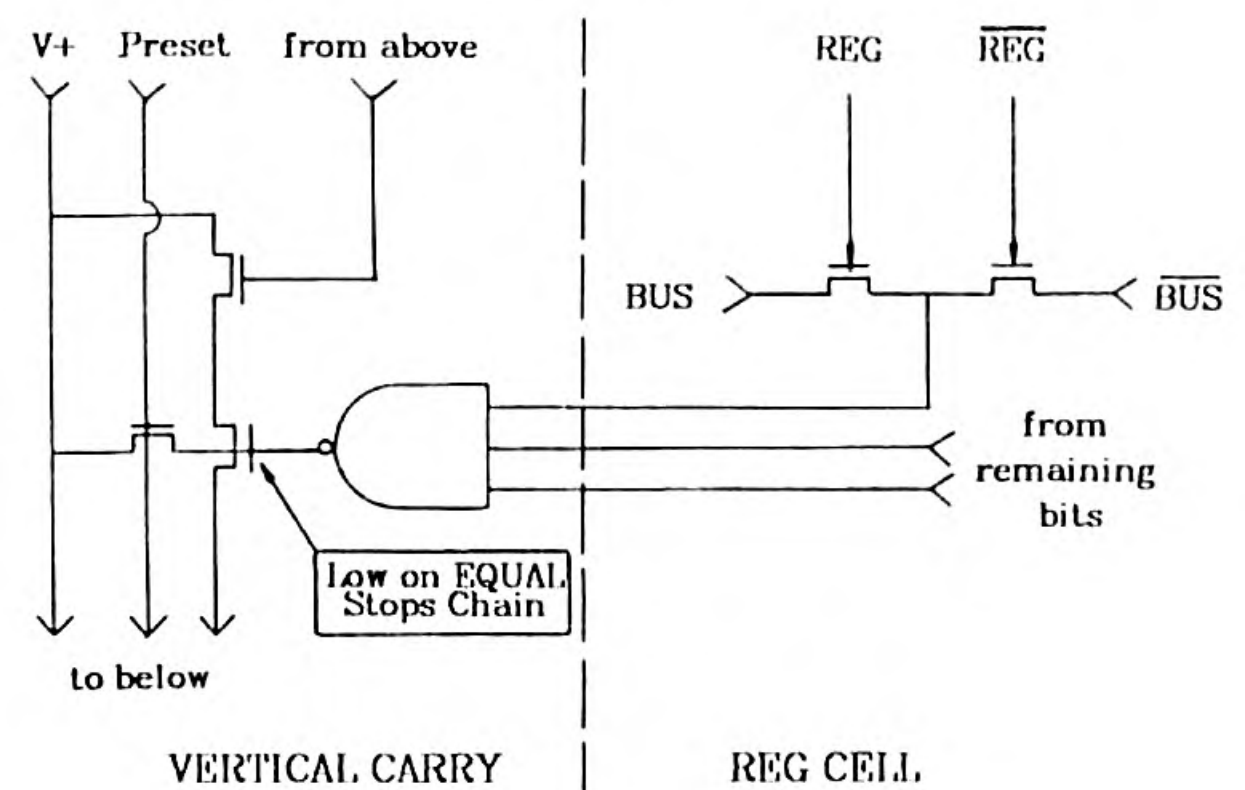


Figure 6. Sequence Vertical Carry Chain

The Local Control is used to produce the signals telling the registers where to read their update data from. It takes as input the output of the value comparison from itself, the register above and the register below, and the value of the vertical carry chain indicating its position relative to the data to be discarded. Using these equations the local control produces the "read/move" signals to control the updating of the value and sequence registers.

READ FROM ABOVE :=
 SEQUENCE equal BELOW or HERE
 AND REGISTER ABOVE ≤ BUS

READ FROM BELOW :=
 SEQUENCE equal ABOVE or HERE
 AND REGISTER BELOW $\leq$ BUS

READ FROM BUS :=
 (VALUE $\leq$ BUS
 AND REGISTER ABOVE $>$ BUS
 AND SEQUENCE equal BELOW)
 OR (VALUE $>$ BUS AND
 REGISTER BELOW $>$ BUS AND
 SEQUENCE equal ABOVE or HERE)

The sequence counter is initialized to zero and counts up to $n-1$ (the window size) before resetting to zero. As all of the Sequence Registers are initialized with numbers in vertical descending order, and the Value Registers are initialized to zero, the startup of the filter is correct. Each new value is added at the *greater-than || less-than-or-equal-to* boundary. The new values will be either greater than or equal to zero, so new values will be added above the initial block of zeros. The first value will be placed in the top register, discarding the zero in that register. The next value will be sorted into the top two registers, and so on, until $n-1$ is reached. The discards will now be confined to the top of the register set, so all the movement will remain in a subset of the full register set.

With the Register set updated, there remains only to get the value from the desired ranked value register to the outside world. The Value Bus is precharged to a median value by the Sense Amplifier Circuitry, and then the Rank Selection causes the proper register to begin to change the charge on the output bus. The Sense Amplifier detects and latches this change, and the output buffer makes this available to the outside world.

Recursion can be added to the Filter by the addition of two features. A second Sequence Counter which is loaded initially with $n/2$ and counts the same way as the primary register will provide the sequence counting to identify the old input to be replaced with the output. The output circuit is modified to write back to the Value Bus. With these two changes the circuit becomes recursive with a time penalty of approximately a factor of two.

The Critical Path

Fig. 7 shows the timing of each of the major steps. The upper portion indicates the critical path for the Sequence Numbers, the middle portion is the Value input and comparison, and the lower part concerns the output. It can be seen that a major portion of the time involved is

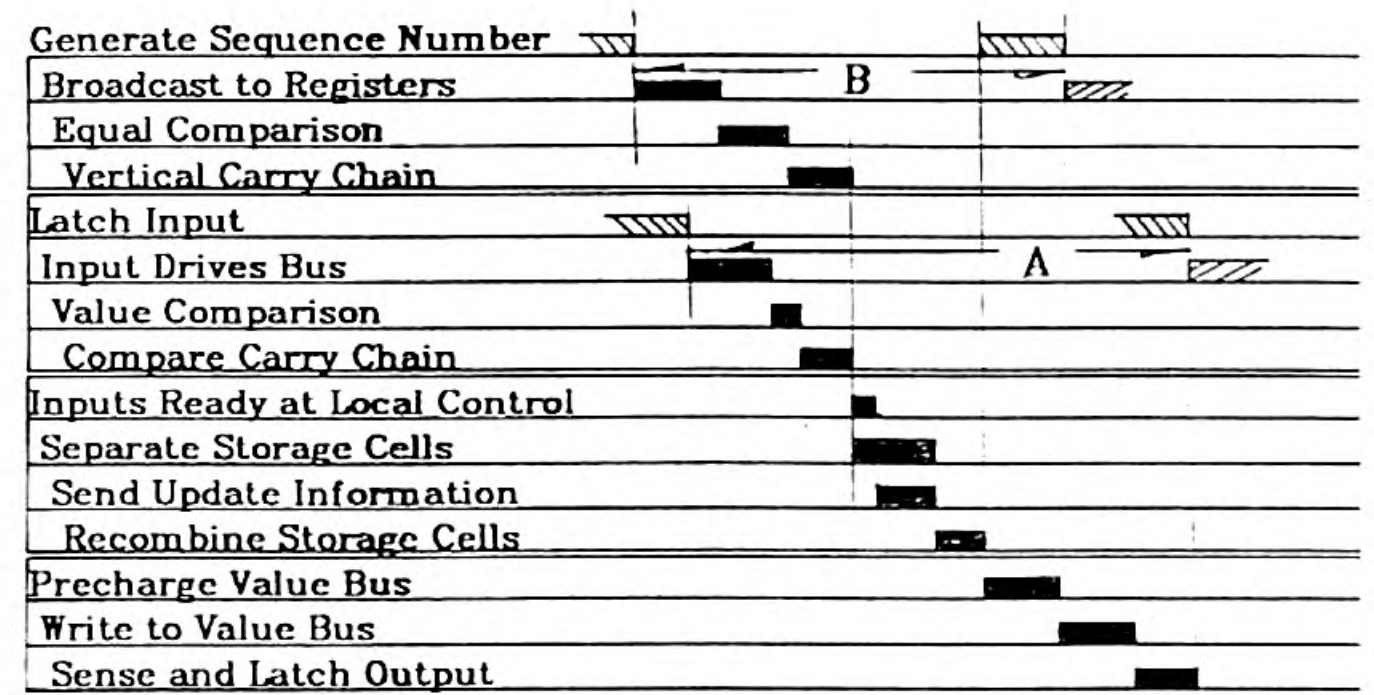


Figure 7. Critical Path Timing of the Update Sequence

taken on the Value bus. A separate output bus may prove in a later version of the chip to make an improvement in the timing. The minimum cycle time is $MAX(A,B)$, where A is the Sequence Cycle Time and B is the Value Cycle Time. Specific times for each of these cycles for different window sizes are discussed in the next section.

SIZE AND SPEED CONSIDERATIONS

The chip, with the architecture of the median filter presented here, has been designed to be implemented with an NMOS process with $\lambda = 2 \mu m$ (feature size = $4 \mu m$). The design rules followed in this project and the values for resistances, capacitances, and sizes, are those established in Mead and Conway[5]. Only one level of metalization is used in the original design.

The overall size of the chip is 6.4 mm by 4.4 mm for a window size of 15. The different components of this area can be seen in the floor plan of the chip Fig. 3. At the center of the chip is the circuitry that implements the median or rank process. This circuit is composed by a block that is repeated a window size number of times. This block is seen in more detail at the bottom of Fig. 3.

The area of the chip can be represented as:

$$Area = A + Fm + Bn[C(m + \alpha(n)) + D] \quad (1)$$

where

$D = D_1 + D_2 + D_3$ (see detail fig.8)

$\alpha(n) = \text{integer}(\log_2(n-1) + 1)$

n : window size

m : number of input bits

A : area of, internal clock, clock inputs, control inputs, sequential register

F : area of, input/output buffer, line driver, (per input bit)

B : height of the block (height of the tallest element)

C : width of the value register and sequential register

D : width of rank latch, end terminations, local control and vertical carry chain

$\alpha(n)$ is the number of bits needed by the sequential register as function of the window size.

From equation (1) we can see that the area is composed of two constant terms A and F times m ; plus the third term which express the increase in area as a function of the window size. Then equation (1) can be expressed as

$$Area = A + Fm + T(n) \quad (2)$$

where

$$T(n) = Bn[C(m + \alpha(n)) + D] \quad (3)$$

If we make a reasonable assumption respect to the width occupied by the term D we came out with

$$D = 5C$$

and also if we assume an 8 bit input, equation (3) becomes

$$T(n) = BCn(13 + \alpha(n)) \quad (4)$$

In order to compare this implementation with others in the literature, e.g. the one presented by Oflazer [4], we compare the expression of equation (4) with one that grows as the square of the window size. To make a reasonable comparison, we must compare the areas of both implementations for the same window size. Table 1 shows values for the different areas, expressed in mm^2 , $\lambda = 2 \mu m$, $B = 100 \lambda$, and $C = 70 \lambda$.

AREA in mm^2 vs. n				
	n	$A + Fm$	$T(n)$	Area
Udel	15	20	8	28
work	5	20	3	23
Oflazer[4]	5	10	10	20

Table 1. Area vs. Window Size

It must be pointed out that in the case of the Udel work the area ($A + Fm$), has been kept constant when changing the window size from 15 to 5. Also the values used for the Oflazer implementation are estimated from his paper, assuming an active sorting area of 50 percent of the chip. These curves are presented in Fig. 8. From this figure we can conclude that, taking in account the assumptions made here, both implementations are comparable in size up to a window size of about 7, for small window sizes, the chip area is dominated by the constant terms, and that for large window sizes the

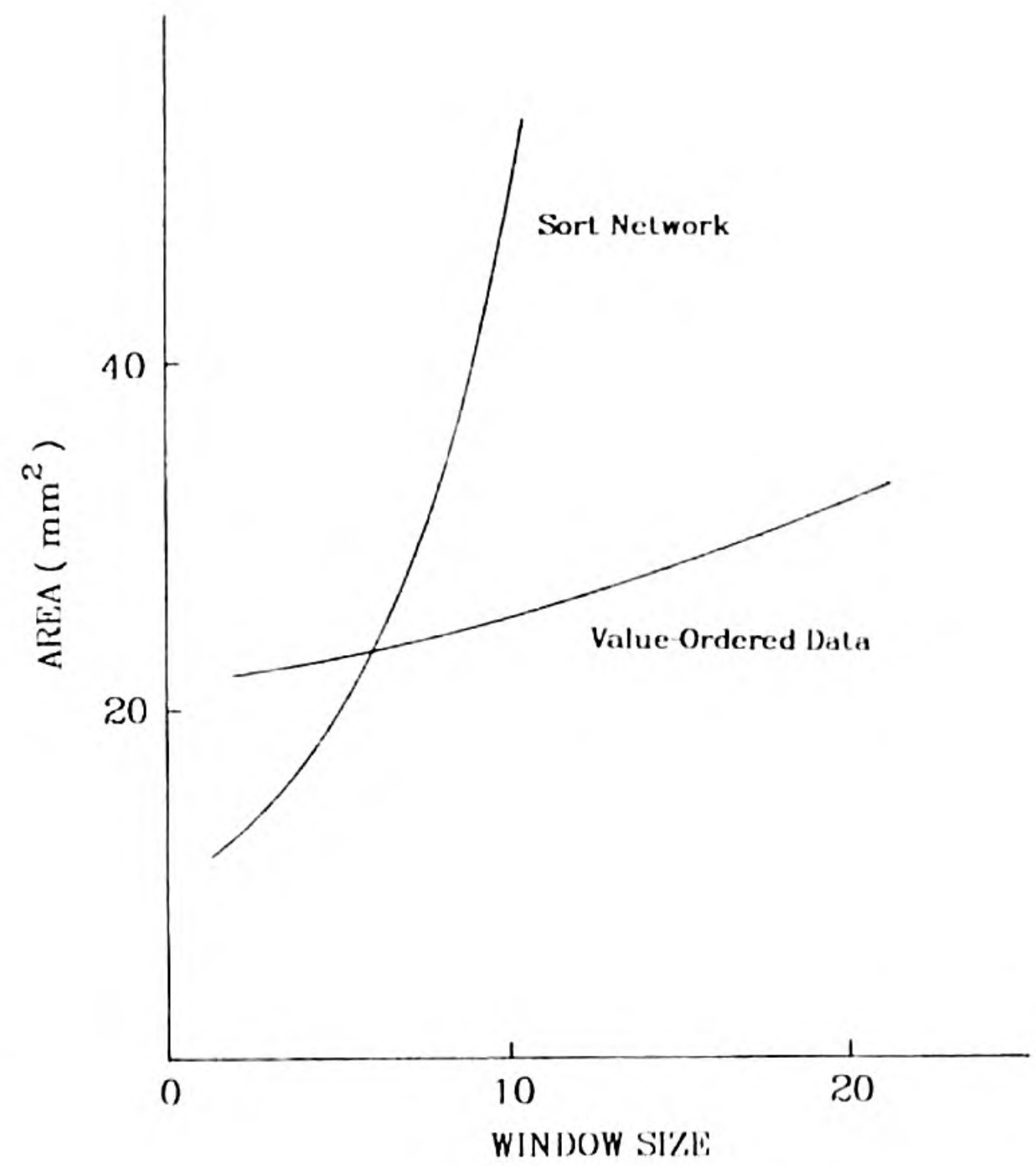


Figure 8. Area vs. Window Size

Sorted-List Implementation is much smaller. It should be noted that Oflazer recognizes this inherent size disadvantage in his paper.

SPEED

The speed of this circuit, has been calculated using values of resistivity, and capacitance per unit length, of the different levels, from Mead and Conway[5]; arriving at an upper bound on the speed of 40 Mega-samples per second for a window size of 15. This is the optimum value that can be obtained and requires clocks with nanosecond precision. Oflazer indicates an Optimum speed of 50 Mega-samples per second with a window size of 5.

The main constrains that limit the speed of this system are : the capacitance of the metal lines coupled with the resistance of the polysilicon lines used as cross-overs from horizontal distribution busses to the vertical busses in the Register Set. This is exemplified in Table 2. We can see that as the area of the chip increases, i.e. bigger window size, all the times related to drive, precharge, and read from the bus are strongly affected. This problem can be partially overcome if we were to use a second level of metalization, or even a low resistance super-polysilicon. This would make it worth decreasing the output resistance of the bus drivers to increase the charging/discharging speed of the busses.

TIMING [in nsec]			
	WINDOW SIZE		
	8	16	32
Latch Input	-	-	-
Input Drives Bus	1.1	1.8	3.6
Compare and Carry	1.0	1.0	1.0
Clock	1.0	1.0	1.0
Propagate	0.6	0.9	1.8
Vertical Carry	1.1	1.8	3.6
Local Control	< 0.1	< 0.1	< 0.1
Clock	1.0	1.0	1.0
Propagate	0.6	0.9	1.8
Control Signals out	4.0	4.0	4.0
Read From Bus	0.1	0.1	0.1
Hold1, Hold2	4.0	6.0	8.0
Precharge Bus	2.0	3.0	6.0
Write to Bus	1.1	1.8	3.6
Clock	1.0	1.0	1.0
TOTAL TIME	18.7	24.4	36.6
Mega-samples/second	53	41	27

Table 2. Timing vs. Window Size

CONCLUSIONS

This chip has the possibility of realizing real time video rates of median filtering. Filtering a 512 by 512 image at 30 frames per

second requires an input/output rate of ~8 mega samples per second. This is within the range of the chip. The chip embodies a design which is both fast and flexible. The window size and the rank output can be programmed "on the fly" making this useful in adaptive filtering applications.

References

- [1] J. W. Tukey, *Exploring Data Analysis*, Reading, Ma.: Addison-Wesley, 1971 (Preliminary edition.)
- [2] G. R. Arce, N. C. Gallager, and T. A. Nodes, "Median Filters: Theory for One and Two Dimensional Filters", *Advances in Computer Vision and Image Processing*, JAI Press, (to be published) 1984.
- [3] G. R. Arce and R. J. Crinon, "Median Filters: Analysis for Two Dimensional Recursively Filtered Signals", *Proceedings of the 1984 Int. Conf. on ASSP*, San Diego, Ca., March 1984.
- [4] K. Oflazer, "Design and Implementation of a One-chip 1-D Median Filter", *IEEE Transactions on Signal Processing*, November 1983.
- [5] C. Mead and L. Conway, *Introduction to VLSI Systems*, Reading, Ma.: Addison-Wesley, 1980.

MEDIAN FILTERS AS SPECTRAL WINDOWS

R. K. Pearson*

G. R. Arce**

*E. I. duPont de Nemours
Experimental Station
Wilmington, DE 19898

**Department of Electrical Eng.
University of Delaware
Newark, DE 19716

Abstract

One problem commonly arising in digital signal processing is the estimation of a signal's power spectrum from a finite number of noisy samples. The presence of noise is a significant complication in practical problems and a variety of "spectral window" techniques have been developed to overcome it. This paper investigates the use of the median filter as an alternative to these techniques when the noise is non-Gaussian. The median filter has been successfully applied to image enhancement problems in the presence of non-Gaussian noise. Numerical results are presented here comparing the performance of the median filter as a nonlinear data window to classical linear data windows for spectral estimation in the presence of both Gaussian and non-Gaussian noise.

I. Introduction

The problem of estimating the power spectral density (PSD) of a signal from a finite number of noisy samples arises in many different fields. Examples include the detection of automotive engine faults from crankshaft position or velocity,¹ the interpretation of radar receiver signals² and ultrasonic diagnosis of vascular disease from skin probe measurements.³ In practice, one of the most common methods of spectral estimation is to compute the square of the magnitude of an N-point fast Fourier transform (FFT) of the available data and divide the result by N. This estimate is called the periodogram and is very popular because of the computational efficiency of the FFT algorithm,⁴ but it exhibits two inherent limitations.⁵ First, because the signal being analyzed is usually modeled as a random quantity, the periodogram is also a random quantity and must be characterized statistically. An analysis of the variance of the periodogram reveals that it is a statistically inconsistent estimator of the true power spectral density and must be modified if reliable estimates are to be obtained. Secondly, because it is based on

only a finite set of discrete signal samples, the periodogram is a biased spectral estimator whose expectation is given by the convolution of the true PSD with a spectral window function. The bias problem limits the resolution of the spectral estimate (i. e., the ability to distinguish closely spaced features in the spectrum) and is typically overcome by making N large enough to clearly reveal the main features of interest. The variance problem is more severe and is typically overcome by deliberate application of a spectral window that increases bias but reduces the variance to achieve a statistically consistent spectral estimate.

To assess the statistical characteristics of these spectral estimators, the signals involved are usually assumed to be stationary Gaussian random processes. Recently, however, considerable interest has developed in the use of non-Gaussian random processes to describe experimental data.^{6,7} In such cases, standard static feature estimators for Gaussian random variables may perform very poorly when applied to non-Gaussian data. For example, although the arithmetic mean is the optimal estimator of central tendency for Gaussian data, it is very sensitive to deviations from normality.⁷ This paper examines the impact of such deviations on the spectral estimation problem where the effects should be even greater. In particular, statistical outliers in the input data sequence can greatly increase the apparent bandwidth of the underlying physical process by introducing non-physical high-frequency features in the estimated spectrum. To overcome this problem, we investigate the use of median filters as nonlinear spectral windows to reduce the effects of heavy-tailed non-Gaussian noise corrupting the measurements of an inherently smooth physical signal. Median filters have been advocated for data smoothing by J. W. Tukey⁸ and have been applied successfully in such image processing applications as edge enhancement⁹ and multispectral classification.¹⁰ As a specific example, this paper develops

spectral estimates for a deterministic Gaussian pulse corrupted by zero-mean, stationary white noise samples drawn from a heavy-tailed non-Gaussian distribution. These results are compared with those for Gaussian observation noise and with similar results obtained using classical linear spectral windows.

II. Classical Spectral Estimation

The power spectral density $S_{xx}(\omega)$ of a random sequence $\{x_k\}$ is defined as the discrete Fourier transform (DFT) of the autocorrelation function $R_{xx}(k)$, i.e.

$$S_{xx}(\omega) = \sum_{k=-\infty}^{\infty} R_{xx}(k) e^{-jk\omega} \quad (1)$$

where

$$R_{xx}(k) = E\{x_n x_{n+k}^*\} \quad (2)$$

Here, $E\{f(x)\}$ represents the expectation of the random quantity $f(x)$ with respect to the probability distribution of $\{x_k\}$. Note that both the autocorrelation and the PSD are deterministic but unknown, and that neither of them can be computed exactly without a complete statistical description of the data sequence $\{x_k\}$. In practice, however, this statistical description is not known and the PSD must be estimated from the available samples of $\{x_k\}$. One such estimate is the periodogram $P_N(\omega)$, defined as

$$P_N(\omega) = (1/N) \left| \sum_{k=0}^{N-1} x_k e^{jk\omega} \right|^2 \quad (3)$$

The bias of the periodogram may be determined by considering the N data samples appearing in the FFT sum as an infinite sequence $\{y_k\}$ obtained by multiplying the data sequence $\{x_k\}$ by a weighting function whose value is 1 for $k=0,1,\dots,N-1$ and 0 otherwise. It is then straightforward to show that the expected value of the periodogram is the true power spectral density convolved with the discrete Fourier transform of the weighting sequence. Physically, this introduces a broadening of the estimated PSD that limits spectral resolution and can only be overcome by either increasing N or employing extrapolative techniques like the maximum entropy (autoregressive) method.⁵

The variance of the periodogram depends on fourth moment expectations of the form $E(x_k x_l x_m x_n)$ which can be reduced to a sum of products of autocorrelations if $\{x_k\}$ is a Gaussian sequence.⁴ Specifically it can be shown in this case that the variance of the periodogram is bounded below by the square of its expectation. Thus, if the periodogram is viewed as a random signal, its signal-to-noise ratio is on the order of unity or less. Further, this result implies that the variance of the periodogram does not approach zero with increasing N , making it an inconsistent estimator of the power spectral density. The classical solution to this problem is to impose an intentional spectral window that reduces variance at the expense of increased bias. These techniques apply either multiplicative

weights or discrete convolution to the original data sequence, the estimated autocorrelation sequence or the resulting periodogram to arrive at a new estimate. Note that this suggests six distinct possible meanings for the term "spectral window", a fact that can lead to considerable confusion. Here, we are concerned with "data windows"⁴ that are applied directly to the data sequence $\{x_k\}$, either as multiplicative weightings or as discrete convolutions.¹¹

III. Non-Gaussian Statistics

Scalar physical measurements $\{x_k\}$ are usually modeled as stationary random processes with jointly Gaussian statistics and unknown mean, variance and autocorrelations. For example, a common model is

$$x_k = s_k + n_k \quad (4)$$

where $\{s_k\}$ is a deterministic sequence describing the underlying physical behavior of interest and $\{n_k\}$ is an independent, identically distributed sequence of zero-mean Gaussian random variables (i.e., white Gaussian noise) describing the measurement errors. For such a model, it is frequently argued that the measurement errors represent a sum of independent contributions from different sources so that $\{n_k\}$ should be normally distributed by the central limit theorem.¹² However, experimental evidence suggests that this argument is not always valid⁶ and results from the field of robust statistics illustrate that such deviations from normality can have significant consequences.⁷ In particular, the presence of outliers (i.e., points more than a few standard deviations from the mean) in the data can lead to large errors in estimates derived under the assumption of normality. In practice, outliers can arise from either gross measurement errors (e.g., sensor failures in a data acquisition system) or random physical phenomena that are described by "heavy-tailed" probability distributions.

One of the most extreme examples of such a distribution is the Cauchy distribution:

$$p(x) = \frac{1/\pi}{1+x^2} \quad (5)$$

which decays so slowly that its variance is infinite, as is the expectation of the absolute value of x . Consequently, samples drawn from a Cauchy distribution exhibit enormous variation about the mean. Less drastic heavy-tailed distributions can be obtained by considering Student's t -distribution as a symmetric, zero-mean non-Gaussian family indexed by the degrees

of freedom parameter.⁷ This family includes everything from the Cauchy distribution (Student's t-distribution with infinite degrees of freedom). Intermediate members of the family have heavier tails than the Gaussian distribution but are better behaved than the Cauchy distribution. In particular, all t-distributions with three or more degrees of freedom exhibit finite variances.

As noted in the introduction, the mean tends to be a poor estimator of the center of a distribution in the presence of outliers. Intuitively, it seems likely that the presence of outliers would have an even greater impact on spectral estimates since they introduce unphysical sharp discontinuities in the data, creating high frequency spectral features that should not be present. Also, recall that the variance of the periodogram depends on the joint fourth moments of the data sequence $\{x_k\}$. This suggests that the variance may be even larger for heavy-tailed non-Gaussian distributions since their fourth moments are correspondingly larger than for Gaussian sequences (e.g., they are leptokurtic¹²). A better estimator of central tendency for very heavy-tailed distributions is the median of the measured data samples.⁷ This suggests that median filters may be better suited to spectral estimation in the presence of outliers than classical convolution-based windows (since convolutions may be viewed as weighted averages) or multiplicative spectral windows (which may be viewed as convolutions in the appropriate transform domain).

IV. The Median Filter

The term median filtering is used by us as a generic term for a class of non-linear digital filtering operations. The basic concept is very simple. A window containing an odd number of sample points is passed over a sampled signal. At every window position, as it is moved across the signal, the sample values within the window are ranked in value. The filter output, which is associated with the sample location at window center is the median or mid value of these ranked samples. Actually, the output value could be set to any ranked value, not just the median value; however, the median value is the one most frequently used. The median filter concept can be applied to signals of one, two, or even higher dimensions. There are 2 types of median filter operations; their description is in order.

1) The standard median filter.

Median filtering is a discrete time process. We slide a $2N + 1$ points wide window across an entire input signal. At

each step, we compute the median value of the $2N + 1$ points spanned by the window. At both ends of the signal, N points are appended to the beginning and end of the signal to account for start up and end effects. The value of the front end points is equal to the value of the first point of the signal, and the value of the last end points is equal to the value of the last point of the signal. If $\{x(\)\}$ and $\{y(\)\}$ are the input and the output sequences respectively, then the output is:

$$y(a) = \text{median}[x(a-N), \dots, x(a), \dots, x(a+N)] \quad (1)$$

This operation is illustrated in Fig. 1a

2) The recursive median filter

The recursive median filter is a modification of the standard filter where the value of the center sample of the window is replaced by the computed median value before the window is shifted to the next position. In other words, the filtered sequence is

$$y(a) = \text{median} [y(a-N), \dots, y(a-1), x(a), x(a-1), \dots, x(a-N)] \quad (2)$$

The advantages of the recursive median filter over the standard will be mentioned after we define the following signal structures.

- A Constant neighborhood is a region of at least $N+1$ consecutive identically valued points.
- An Edge is a monotonically rising or falling set of points, whose first and last points are part of two constant neighborhoods (each of different value).
- An Impulse is a set of no more than N consecutive points whose value is different from the two surrounding regions which are two constant neighborhood identically valued.
- A Root is an invariant signal to further median filtering process.

It has been shown that at most $(L-2)/2$ passes of the median filter are required to reduce an L length signal to its root. The advantage of the recursive operation is that the root signal is obtained after the first pass of the recursive filter, hence reducing processing time. This is desirable if real time processing is needed. Although the roots generated by the standard and recursive filter may differ for the same input signal, they have the same characteristics. Figure 1 illustrates the two processes.

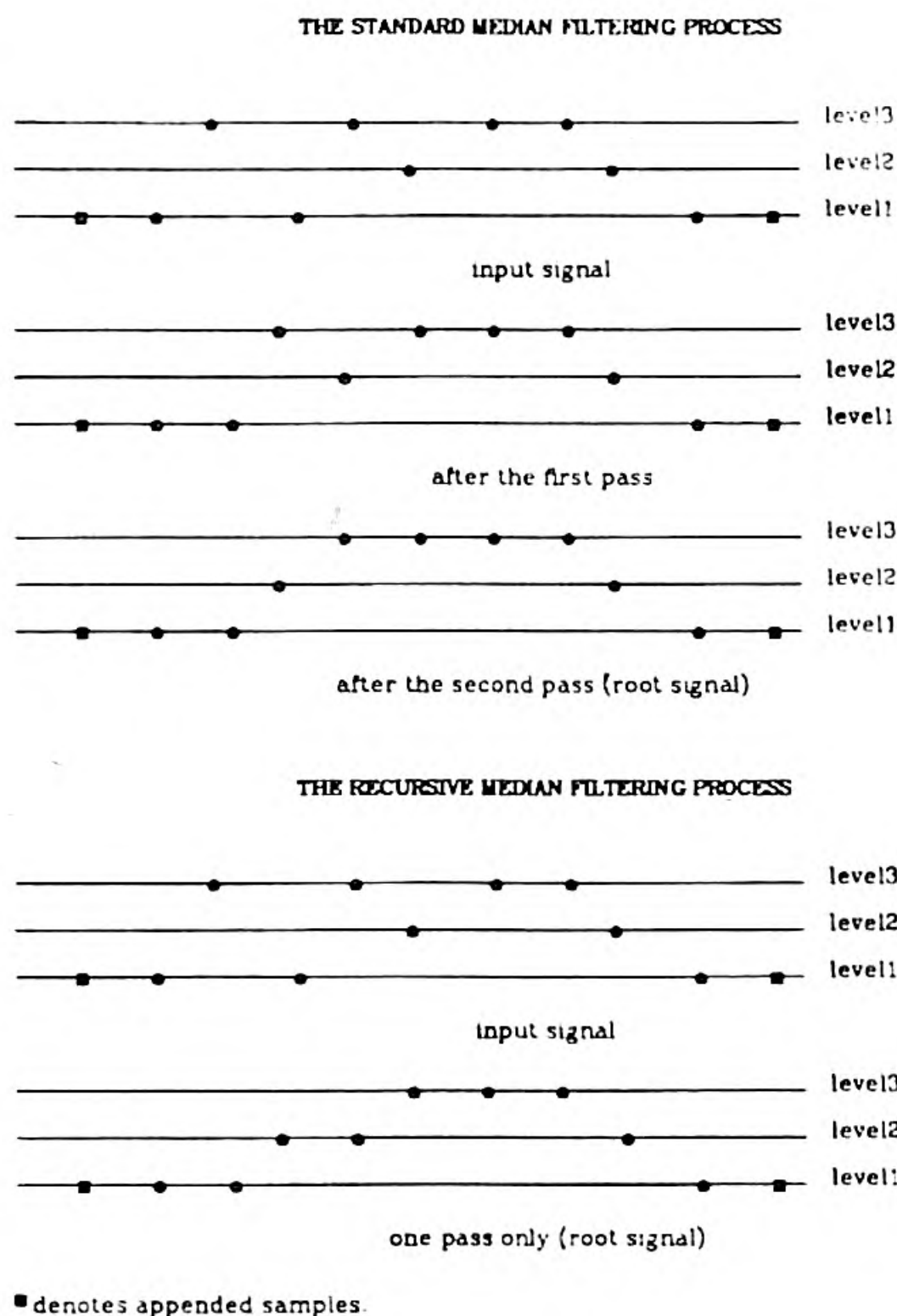


Fig. 1. Signal Filtered by Both Types of Median Filters.

V. A Numerical Example

To illustrate the concepts just described, consider the signal model of equation (4) in which (s_k) is the Gaussian pulse sequence

$$s_k = 10 \exp(-(k-200)^2/200) \quad (6)$$

The noise sequence (n_k) will be taken as discrete samples of a zero mean white stationary process with either one of two symmetric distributions. To establish a standard for comparison, the first model will assume (n_k) to be distributed normally with unit variance. The second, non-Gaussian, noise model considered will assume samples are drawn from Student's t -distribution with three degrees of freedom. This results in a reasonably heavy-tailed noise distribution with finite variance that will be referred to as t -noise for simplicity.

Spectral estimates for the data sequence (x_k) will be derived from the 1024 point subsequence $x_1, x_2, \dots, x_{1024}$. In this case, the sample size N is large enough that periodogram truncation effects are negligible (i.e., s_1 and s_{1024} are

both less than the smallest constant representable in the single precision floating point computations considered here). This is illustrated in figure 2, showing the 1024 point periodogram of the noise-free sequence (s_k) , evaluated via the FFT at the discrete frequencies $2\pi n/N$ and plotted as a function of the frequency index n . Since the Fourier transform of a Gaussian pulse is itself Gaussian, it is clear that finite sample size effects are minimal (e.g., there are no spurious high frequency spectral ripples characteristic of truncation effects).

Figure 3 demonstrates the effects of Gaussian noise on the unwindowed periodogram. The main spectral lobe centered at zero is still apparent, but is somewhat obscured by the presence of many narrow spikes. Even more significant, however, is the presence of high frequency spectral features due entirely to the noise. Both of these problems are even more evident in figure 4, which shows the effects of t -noise on the unwindowed periodogram. Here, while the main spectral lobe is still visible, it is much more fragmented than in figure 3 and the spurious high frequency spectral features are very pronounced. In fact, the spurious peaks at $n \sim 200$, $n \sim 340$, $n \sim 430$ and $n \sim 500$ are so pronounced that they might be interpreted as nonexistent high frequency components in the signal (s_k) .

To overcome the effects of noise, we consider three candidate spectral windows - the von Hann multiplicative window, a three point linear smoother described by Hamming¹¹ and a three point median filter. The von Hann window (also known as a Hanning window or raised cosine window) is the multiplicative weighting function defined by^{4,11}

$$w_k = 1 + \cos[(2k-N-1)\pi/N] \quad (7)$$

for $k = 1, 2, \dots, N$. Applied directly to the data as data windows (rather than to the autocorrelation lags as lag windows), such weighting functions may be interpreted in either of two ways. First, they may be viewed as a remedy for the discontinuous transitions of the truncated data sequence (x_k) to zero outside the interval $k = 1, 2, \dots, N$. These transitions are a direct consequence of using the periodogram as a spectral estimator and can introduce high frequency spectral distortions. As noted above, however, this is not a concern here because the signal is effectively zero outside the range of available samples. Alternatively, it may be noted that multiplicative data windows (w_k) generate equivalent multi-lag windows given by the convolution

$(w_k * w_{-k})$ which are in turn equivalent to linear smoothing filters applied to the periodogram directly in the frequency domain.

Unfortunately, this approach is poorly suited to the present problem, as illustrated in the windowed spectral estimate of figure 5. Here, because the peak in the weighting sequence (w_k) does not coincide with the peak of the Gaussian pulse (s_k) , the principal impact of the data window on the signal is to reduce its amplitude by about 50%. Further, because the narrow Gaussian pulse (s_k) is centered at sample 200, the portion of the signal (x_k) around $k = 512$ that is most heavily weighted is essentially all noise. Consequently, the spectral estimate obtained with the von Hann window is very poor in this case, both because the reduced pulse amplitude results in a correspondingly reduced main lobe in the spectrum and because noise-induced high frequency spectral features are emphasized by the lower effective signal-to-noise ratio of $(w_k x_k)$ relative to (x_k) . This problem is not specific to the von Hann window, but illustrates the importance of matching any multiplicative data window to the underlying signal (s_k) . For example, this difficulty could be overcome by centering the data window on the Gaussian peak in the measured signal (x_k) , but the other two techniques described below are considerably easier to apply.

In his discussion of windows, Hamming¹¹ proposes the following smoothing filter as a multiplicative frequency-domain window:

$$y_k = (1/4)x_k + (1/2)x_{k-1} + (1/4)x_{k-2} \quad (8)$$

This particular filter is proposed because its frequency response $H(\omega)$ satisfies the conditions $H(0) = 1$ and $H(\pi) = 0$ so that low frequencies are passed unattenuated and high frequencies are essentially eliminated. This is illustrated in figures 6 and 7 where the smoothing filter of equation (8) is applied to the Gaussian pulses contaminated by Gaussian noise and t-noise, respectively. As expected, the low frequency portion of the periodogram (i.e., n between 0 and 100) is essentially the same as that of the unwindowed periodogram while the spurious high frequency peaks are entirely eliminated (i.e., for n above ~ 300).

Finally, figures 8 and 9 show the periodograms obtained from median filtered data. The results obtained for Gaussian noise, shown in figure 8, are essentially the same as those shown in figure 6 for the linear smoothing filter. Figure 9 shows the results obtained for t-noise and these are also similar to those obtained from the linear smoothing filter, but here the differences are more significant. For example, a careful comparison of figures 7 and 9 reveals that there is less fragmentation of the main peak into narrow spikes with the median filter than with the linear smoother. Similarly, comparison of the periodograms between $n \sim 100$ and $n \sim 350$ show that the median filter attenuates spurious high frequency components slightly better than the linear smoother. Conversely, figure 9 also shows that the median filter can emphasize spurious spectral features, in this case generating a spectral peak at $n \sim 60$ that is enhanced relative to the corresponding peaks in either the linearly smoothed estimate (figure 7) or the original noisy spectrum (figure 4).

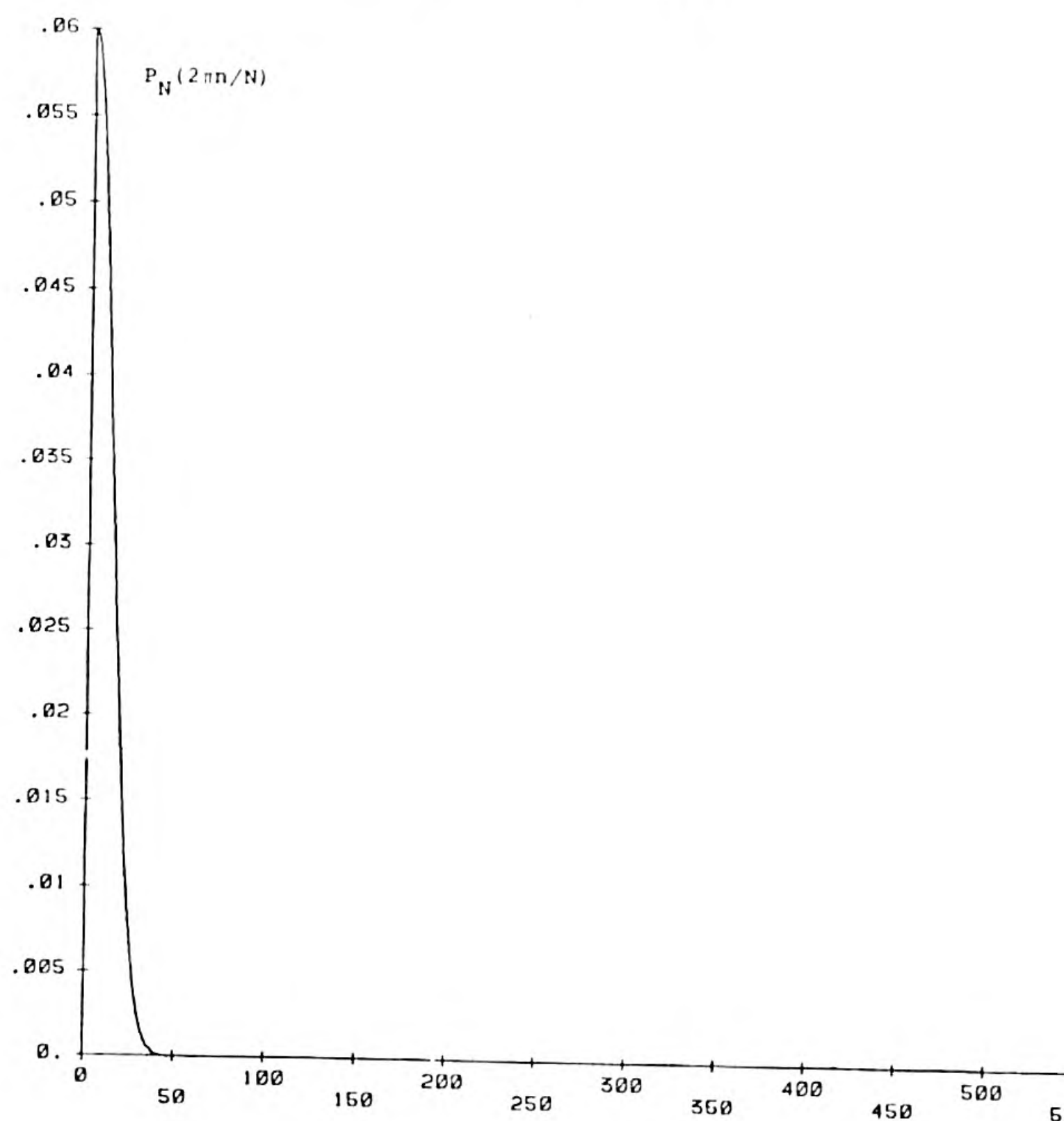


Figure 2 - 1024 point periodogram of a noise-free Gaussian pulse

VI Discussion and Conclusions

The numerical results just presented may be summarized as follows. First, to be useful, multiplicative data windows must be carefully matched to the physically significant information in the observed data sequence $[x_k]$. This may not be possible in the early stages of an investigation, however, as it requires some detailed knowledge about the signal sequence $[s_k]$. Linear smoothers like the one considered here are easier to use since they only require qualitative assumptions on the behavior of the signal's power spectral density. Note, however, that data smoothing with a linear filter corresponds to multiplication of the periodogram by the square of the magnitude of its transfer function. Thus, selection of a linear smoothing filter corresponds to an a priori decision of how to weight the interpretation of the spectral estimate obtained from the data.

In contrast, the median filter performs a data-dependent smoothing of the spectral estimate by imposing something like a local monotonicity requirement on the sequence $[x_k]$. To see this, note that if $[s_k]$ is a monotonic sequence, the median of $2M+1$ successive samples is the $M+1^{\text{th}}$ sample, so the output of a causal $2M+1$ point median filter will be the original data sequence delayed by M samples. Thus, because the Gaussian signal pulse considered here is monotonically increasing for $k < 200$ and monotonically decreasing for $k > 200$, it is almost invariant with respect to median filtering, suffering only a slight flattening of the peak and a one sample delay. Because the median filter is nonlinear, however, it does not follow that the filtered output in the presence of noise will be the superposition of the delayed signal with less intense noise. In fact, this nonlinearity is probably the source of the slightly enhanced spurious peak at $n \sim 60$ in the spectrum of figure 9. This seems likely because the slight clipping effect of the median filter should cause spectral leakage around legitimate features.

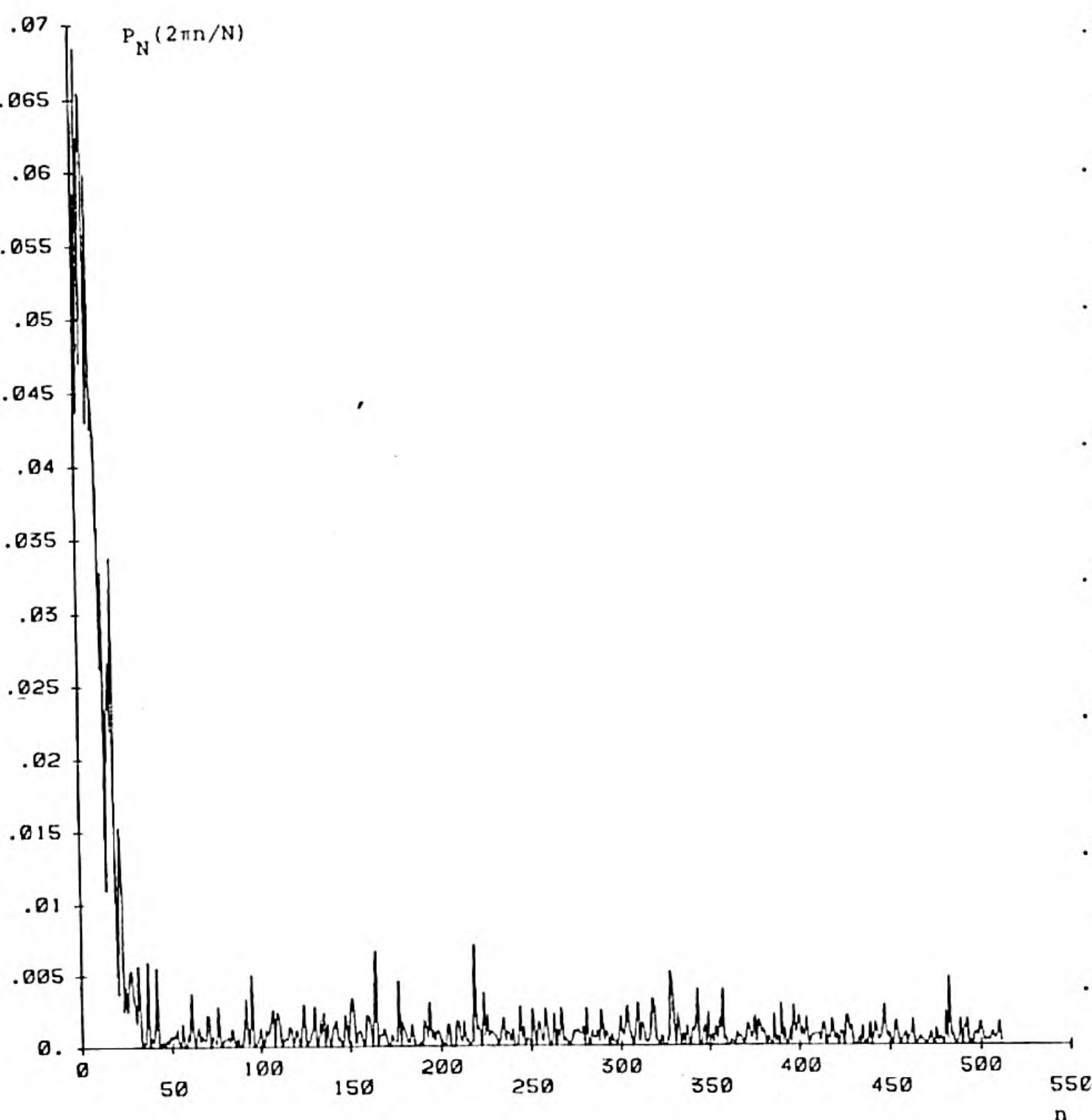


Figure 3 - 1024 point periodogram of a Gaussian pulse in Gaussian noise

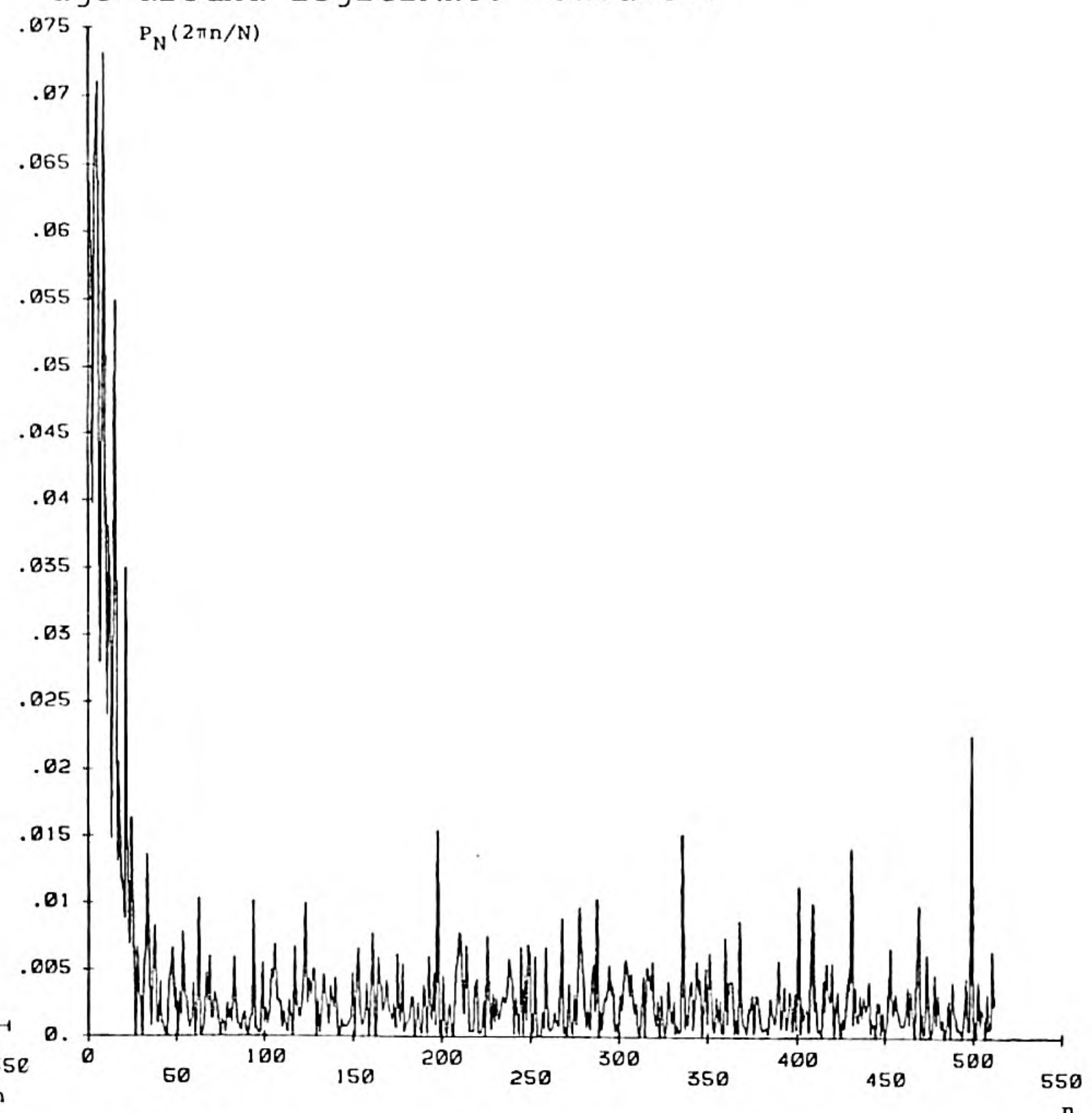


Figure 4 - 1024 point periodogram of a Gaussian pulse in t-noise

The differences between the three point linear smoother considered in the last section and the median filter as data windows are summarized in table 1. Here, comparisons between the unwindowed periodogram, the periodogram obtained with the linear data smoother and the periodogram obtained with the median filter are presented for both Gaussian noise and t-noise. Performance is assessed in terms of a signal-to-noise (S/N) ratio, defined as the ratio of the mean square value of the true PSD to the mean square error between the estimated PSD and the true PSD. From these results, it is clear that the median filter significantly out-performs the linear smoother for the Gaussian pulse contaminated with t-noise. Both filters enhance the quality of the spectral estimate significantly, but the median filter improves the S/N ratio of the estimate by a factor of 4 relative to the unwindowed periodogram while the linear smoother only improves the S/N ratio by a factor of 2. For Gaussian noise, the linear smoother yields the better spectral estimate, but only slightly.

It is not clear how far these results can be generalized, but they do suggest that median filters can be used very successfully as nonlinear data windows for the enhancement of spectral estimates for non-Gaussian random processes. Further, although comparisons were not made against the optimal linear smoother, the results presented here for Gaussian noise also suggest that the efficiency loss incurred in using median filters instead of linear smoothers may be relatively slight. Thus, the median filter may be a very useful data cleaning tool in spectral estimation unless there is reason to believe a priori that the signal probability distribution is platykurtic¹². In such short-tailed distributions (e.g., the uniform distribution), the median is known to be a relatively inefficient estimator of central tendency⁷, suggesting that linear smoothing may be a much better technique for enhancing spectral estimates. It has been suggested, however, that heavy-tailed distributions occur more commonly in practice (c.f., Lehmann⁷, p. 365).

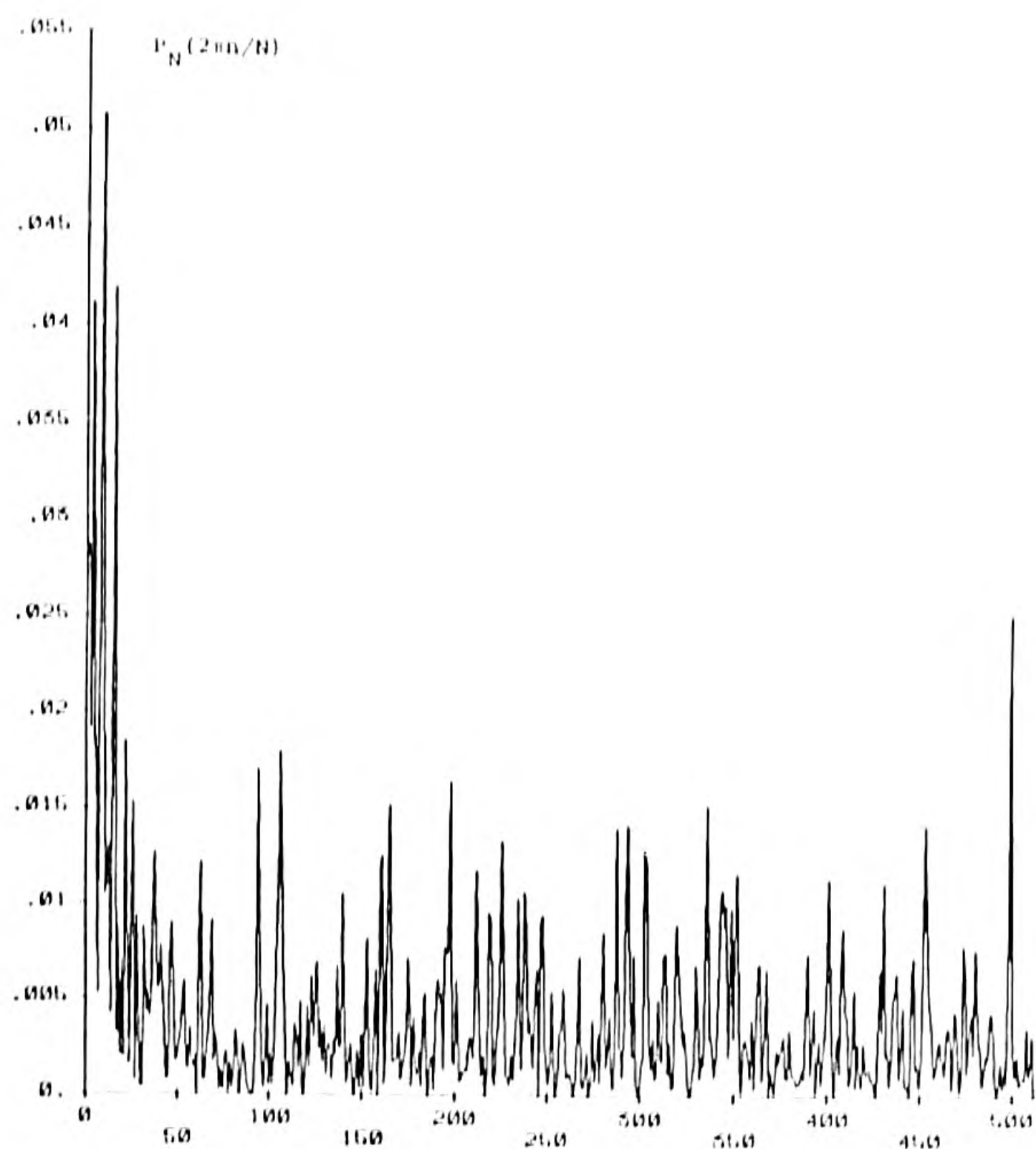


Figure 5 - Effect of a von Hann multiplicative data window on the periodogram of Figure 4

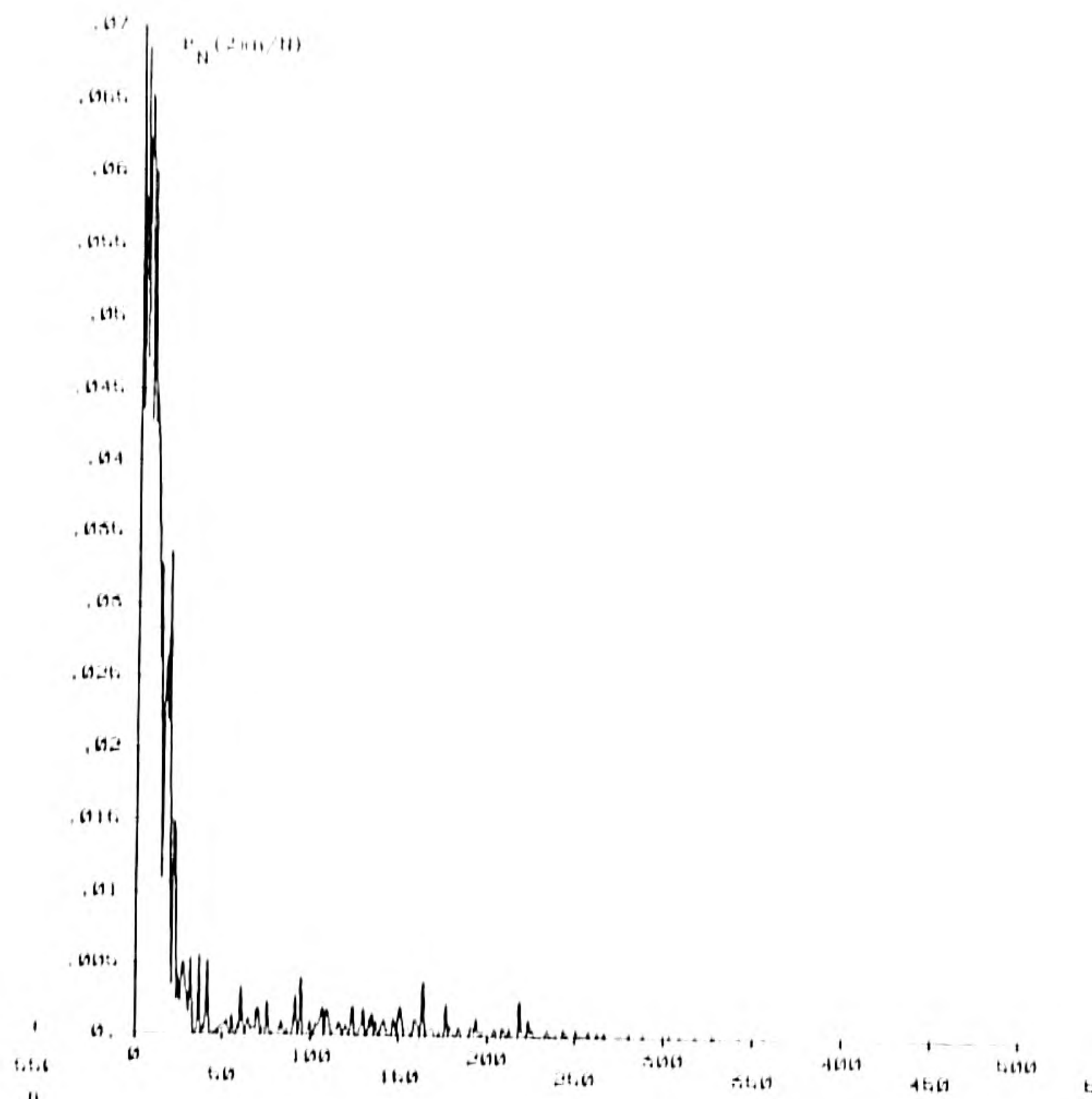


Figure 6 - Effect of 1-point data smoother on the periodogram of Figure 4

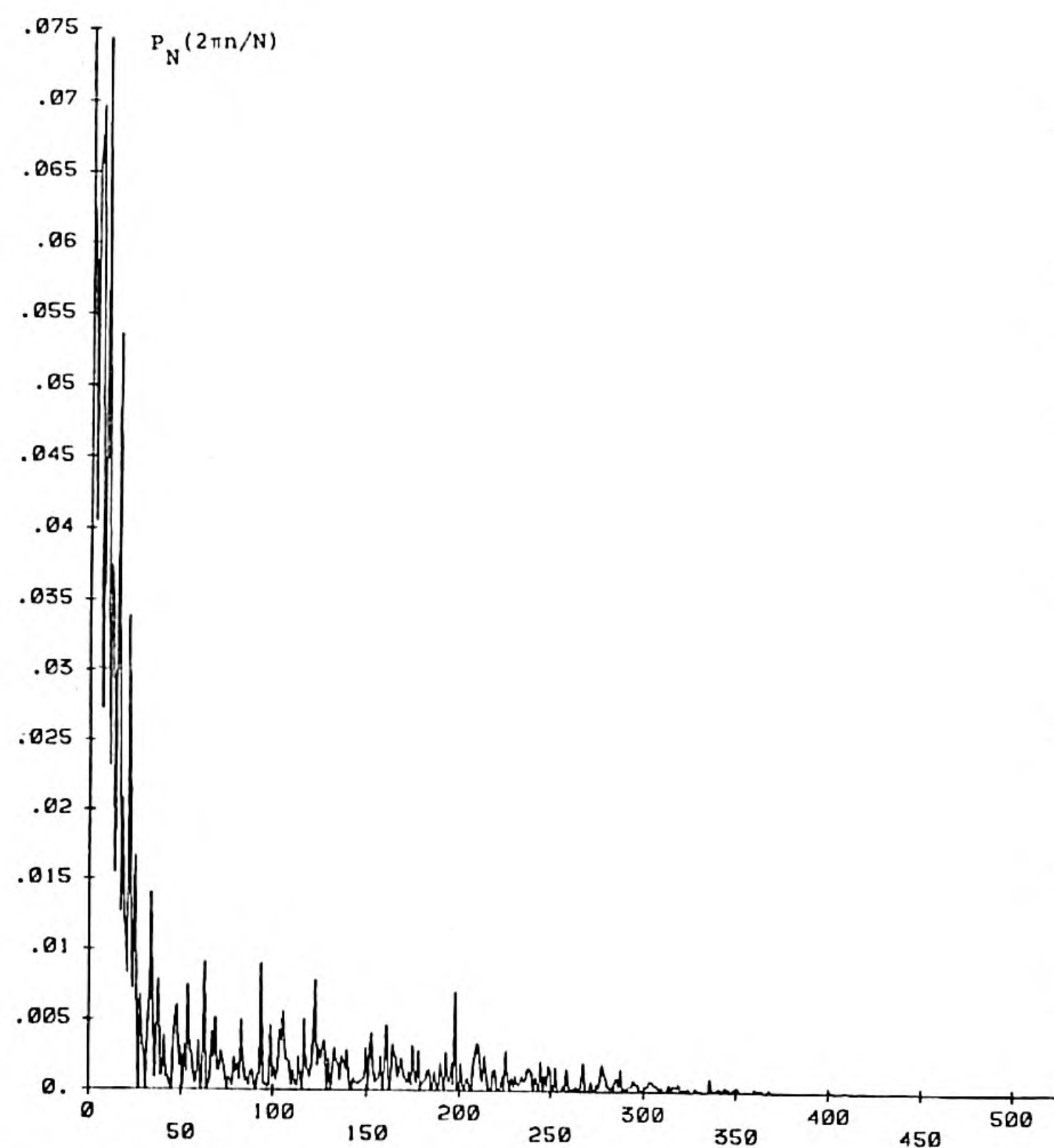


Figure 7 - Effect of 3 point data smoother on the periodogram of Figure 4

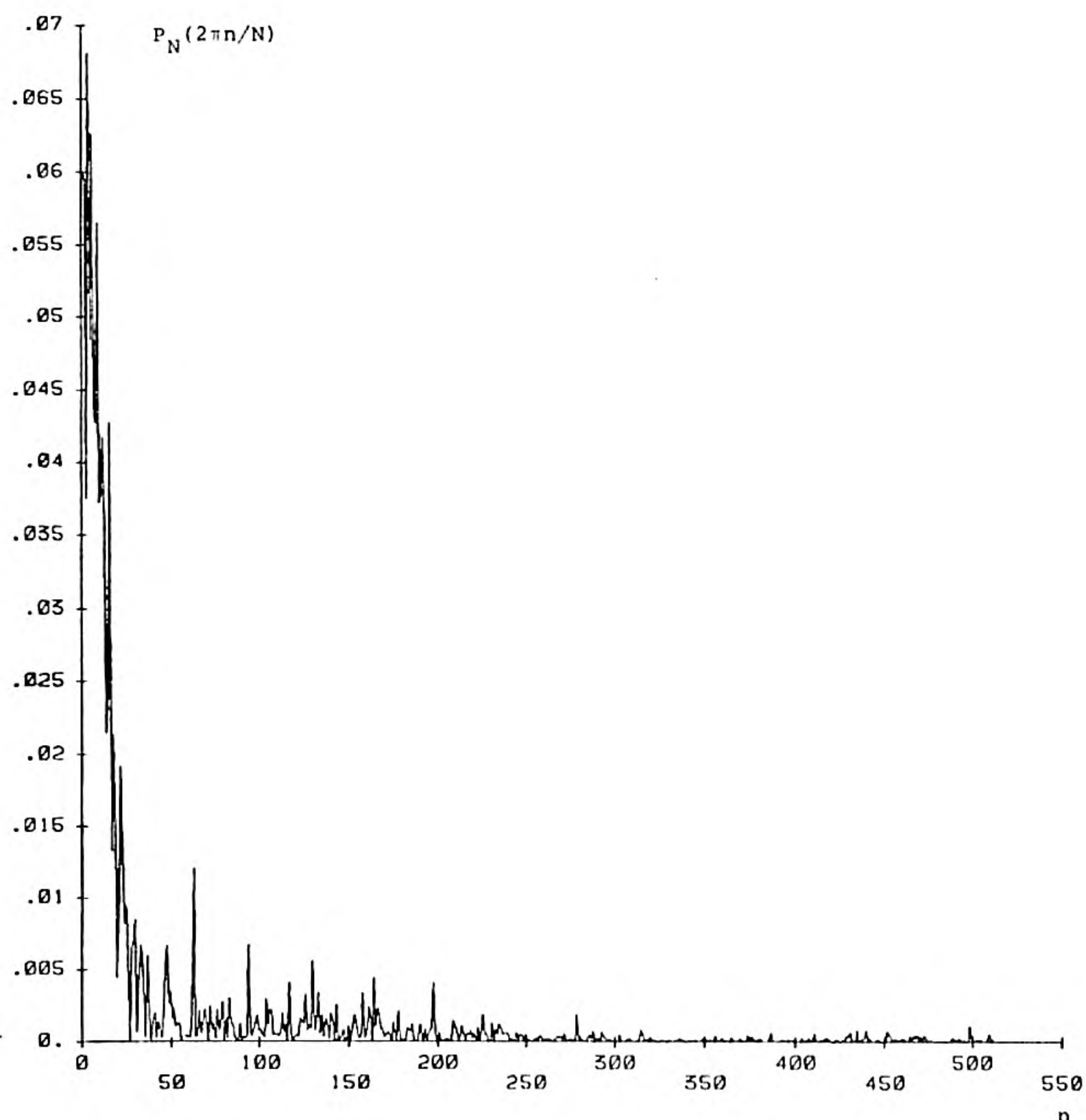


Figure 9 - Effect of 3 point median filter on the periodogram of Figure 4

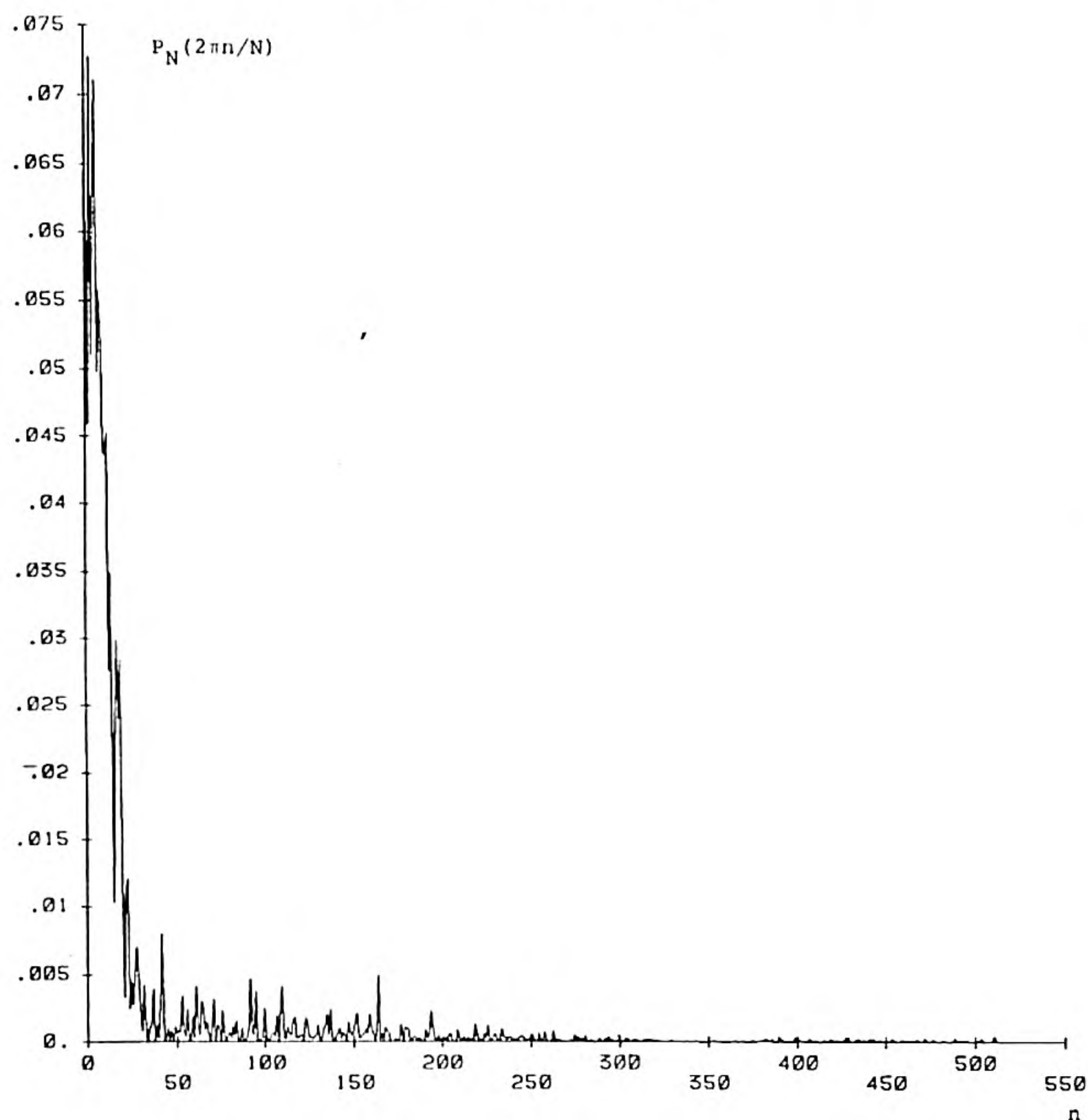


Figure 8 - Effect of 3 point median filter on the periodogram of Figure 3

Table 1
Comparison of Spectral Estimators

Noise Source	Window	S/N Ratio
t-noise	None	3.34
"	Linear smoother	6.22
"	Median filter	14.34
Gaussian	None	13.65
"	Linear smoother	18.32
"	Median filter	17.96

VII References

1. A. K. Sood, A. A. Fahs and N. A. Henein, "A Real-Time Microprocessor-Based System for Engine Deficiency Analysis," IEEE Trans. Industrial Electronics, Vol. IE-30, 1983, pp. 159-163.
2. A. W. Jenney and D. C. Lai, "Detection Characteristics of the Frequency-Domain Radar Receivers," Proceedings of the 1st ASSP Workshop on Spectral Estimation, McMaster University, August 1981.
3. P. Zuech, R. S. C. Cobbold, M. S. Kassam and K. W. Johnston, "Application of Real-Time Spectral Analysis for Doppler Ultrasound Assessment of Vascular Disease," Proceedings of the 1st ASSP Workshop on Spectral Estimation, McMaster University, August 1981.
4. S. A. Tretter, Introduction to Discrete-Time Signal Processing, John Wiley and Sons, New York, 1976.
5. S. M. Kay and S. L. Marple, Jr., "Spectrum Analysis - A Modern Perspective," Proc. IEEE, Vol. 69, 1981, pp. 1380-1419.
6. G. R. Phillips and E. M. Eyring, "Comparison of Conventional and Robust Regression Analysis of Chemical Data," Anal. Chem., Vol. 55, 1983, pp. 1134-1138.
7. E. L. Lehmann, Theory of Point Estimation, John Wiley & Sons, New York, 1983.
8. J. W. Tukey, Exploratory Data Analysis, Addison-Wesley, Reading, Mass. 1977.
9. G. R. Arce and N. C. Gallagher, "Stochastic Analysis for the Recursive Median Filter Process," To appear in the IEEE Transactions on Information Theory.
10. G. R. Arce, E. Malaret and D. A. Landgrebe, "Multispectral Classification of Noisy Data Using Median Filters," Proceedings of the 6th Micro-Delcon, University of Delaware, 1983.
11. R. W. Hamming, Digital Filters, Prentice-Hall, Englewood-Cliffs, New Jersey, 1983.
12. G. E. P. Box and G. C. Tiao, Bayesian Inference in Statistical Analysis, Addison-Wesley, Reading, Mass., 1973.

PROCEEDINGS SEVENTH ANNUAL
MICRO-DELCON '84

ISBN 0-8186-0554-5
IEEE CATALOG NO. 84CH2004-0
LIBRARY OF CONGRESS NO. 83-83444
IEEE COMPUTER SOCIETY ORDER NO.554